Design, Modeling, and VLSI Implementation of a RISC Dataflow Array Processor

by

Aamir Alam Farooqui

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

ELECTRICAL ENGINEERING

May, 1995

Design, Modeling, and VLSI Implementation of a RISC Dataflow Array Processor

BY

Aamir Alam Farooqui

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In
Electrical Engineering

May, 1995

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN, SAUDI ARABIA
COLLEGE OF GRADUATE STUDIES

This thesis, written by

Aamir Alam Farooqui

under the direction of his Thesis Advisor, and approved by his Thesis committee, has been presented to and accepted by the Dean, College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGG.

Thesis Committee :

Dr. Gerhard H. Beckhoff (Chairman)

Dr. Sadiq M. Sait (Co-Chairman)

Dr. M. T. Abu-El-Ma'atti (Member)

Dr. Abdul Rahman Al-Ali (Member)

Dr. Hussein Baher (Member)

S.A AL-Baryat
Department Chairman

Dean, College of Graduate Studies

Date: 12-7-95

Design, Modeling, and VLSI Implementation of a RISC Dataflow Array Processor

MS Thesis

Aamir Alam Farooqui

May, 1995

Dedicated to

my Parents

&

my brother and sisters

whose prayers, guidance, and inspiration led to this accomplishment

Acknowledgment

In the name of Allah, Most Gracious, Most Merciful. Read in the name of thy Lord and Cherisher, Who created. Created man from a {*leech-like* } clot. Read and thy Lord is Most Bountiful. He Who taught {the use of} the pen. Taught man that which he knew not. Nay, but man doth transgress all bounds. In that he looketh upon himself as self-sufficient. Verily, to thy Lord is the return {of all}.

(The Holy Quran, Surah 96)

First and foremost, all praise to Allah, *subhanahu-wa-ta'ala*, the Almighty, Who gave me an opportunity, courage and patience to carry out this work. I feel privileged to glorify His name in the sincerest way through this small accomplishment. I seek His mercy, favor, and forgiveness. And I ask Him to accept my little effort.

Acknowledgment is due to King Fahd University of Petroleum & Minerals for providing support to this research work.

I am indebted to my thesis chairman, Dr.Gerhard F. Beckhoff and co-chairman Dr. Sadiq M. Sait for their help and advice. I acknowledge them for their valuable time, encouragement and guidance especially during the early stages of this work and my MS studies. Working with them was indeed a learning experience.

I am grateful to my thesis committee member, Dr. M. T. Abu-El-Ma'atti , Dr. A.

R. Al-Ali and Dr. Hussein Baher for their deep interest, constructive criticism and stimulating discussions during the course of this work.

I am thankful to the department chairman, and other faculty members for their cooperation.

I am thankful to the Dr. Talal O. Halawani, for his guidance and motivation in deciding a thesis topic. He always encourage implementation work. Due to his encouragement I chose to implement this processor in VLSI,

I also acknowledge the support of CCSE network administrator Mr. Suhaib Khan and Mr. Asjad Mumtaz Khan, for providing me with enough disk space and help for the accomplishment of this work.

I acknowledge with gratitude, the affection and encouragement of my friends in electrical and computer engineering especially Sajid, Asif Sayani, Azhar Quddus, Imtiaz, Yonus, Rafiq, Ahsan, Osama Sabih and Nayyar, my house mates and my friends on the campus especially Atif, Asim, Mansoor, Ibrahim, Ayaz, Asim Rauf, Amir Hashmi and Zaka who helped me and provided a wonderful company through out my stay in KFUPM.

Last but not the least, thanks are due to the members of my family for their emotional and moral support throughout my academic career. No personal development could ever take place without the proper guidance of parents. This work is dedicated to my parents for taking pains to fulfill my academic pursuits and shaping my personality. They taught me the fundamental concept of life, "Tough times never last, tough people do".

Contents

List of Tables

List of Figures

Abstract

Name: Aamir Alam Farooqui

Title: Design, Modeling, and VLSI Implementation Of A RISC Data Flow Array Processor

Major Field: Electrical Engineering

Date of Degree: May 1995

In this thesis the design and VLSI implementation of a highly reconfigurable Dataflow RISC Array processor (DF-RISC-A) is presented. This array processor possesses all the features of static and dynamic dataflow models. It can execute arbitrary algorithms (both recursive and regular), in static and dynamic manner. In order to increase the speed and reduce VLSI chip area, a RISC methodology has been adopted.

Each processing element can execute 25-instructions. In order to facilitate maximum communication between PEs, each PE can communicate with its 8 immediate neighbors using the boundary registers/ports, while it can communicate with the non-neighbor PEs and the host using the communication network and the host bus which runs between two alternate rows of PEs. This results in tighter coupling and faster communication among processing elements. Since the topology can be reconfigurable, it is possible to implement any dataflow graph on this processor array.

A 'Global Network Controller' takes care of the communication between PEs and host. It generates control signals for data transfer between host and PE. A PE can communicate with the host only through this communication controller. This network controller is used to interface the processor array with the parallel port of a Personal Computer.

The processor has been modeled at behavioral level in VHDL, and gate level implementation has been done using OASIS Logic3 Silicon compiler. Each processing element requires 4261 CMOS gates with an area of 7512 × 8081 μm^2.

Master of Science Degree
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia
May, 1995

موجز

الاسم: عامر عالم فاروقى

العنوان: التصميم والنمذجة والتنفيذ با ستخدام الدوائر المتكاملة ذات النطاق الواسع جدا لمعالج مصفوفي ذو خاصية التدفق البياني ومن نوع RISC .

مجال التخصص الرئيسي: الهندسة الكهربأية.

تاريخ الشهاده: مايو ١٩٩٥.

في هذه الأطروحة يتم عرض تصميم و تنفيذ فى نوع (VLSI) لمعالج مصفوفي من نوع RISC ذوحاصية التدفق البياني وقابل لاعادة التشكيل ويمتلك هذا المعالج المصفوفي كل خواص النماذج التدفقية الاستاتيكية والدينا ميكية و يستطيع هذا المعالج أن ينفيذ خوارزميات عشوائية (تكرارية و نظامية) بطريقة استاتيكية ودينا ميكية ولقدتم اعتماد طريقه فى نوع RISC لزيادة السرعة ولتقليل مساحة شريحة VLSI يستطيع كل عنصر من عناصر المعالج المصفوفي تنفيذ ٢٥ أمر. ولأجل تأمين عملية اتصال أعظم بين كافة العناصر، فاءن كل عنصر يستطيع الاتصال بثمانية عناصر مجاورة باستخدام البوابات على حدود الشريحة بينما يستطيع هذا العنصرأن يتصل بسائر العناصر وبالحاسب الرئيسى باستخدام شبكة الاتصالات وناقل الحاسب الرئيسى الذي يجري بين صنفين من العناصروهذا يؤدي الى عملية اتصال اوثق واسرع بين العناصر المعالجة.

وبما أن هذة الطوبولوجية ممكن ان يعاد تشكيلى، فأنة من الممكن تنفيذ أي بياني تدفقي باستخدام هذا المعالج المصفوفي. وهناك وحدة تحكم شبكية تقوم بتنظيم الاتصال بين العناصر المعالجة والحاسب الر ئيسي حيث تصدر اشارات تحكم لنقل المعلومات بين كل عنصر والحاسب الرئيسي، بحيث لايستطيع العنصر الاتصال بالحاسب الاعن طريق وحدة التحكم هذه .

وتقوم وحدة التحكم هذه بالربط بين العناصر المعالجة والحاسب الرئسي وقد تم نمذجة هذا المعالج على المستوي الوظيفي باستخدام لغة VHDL ، وقد تم تنفيذة على مستوى البوابات باستخدام المترجم السليكوني فى نوع OASIS Logic3 ويتطلب كل عنصر معالج ٤٢٦١ بوابة CMOS بمساحة تبلغ ٧٥١٢x٨٠٨١ ميكرو متر مربع.

درجة الماجستير في علوم الهندسه
جامعة الملك فهد للبترول والمعادن
الظهران

Chapter 1

Introduction

Digital signal processing, image processing, real-time processing and pattern recognition encompass a variety of mathematical and algorithmic techniques, which are dominated by certain key algebraic methods. These algorithms possess properties such as regularity, recursiveness, and locality, which can be exploited in array processor design. The most popular array processors are systolic arrays and wavefront arrays. These processors are effective in solving highly regular problems such as multiplication, and addition of arrays etc. (they are also called algorithm oriented processors).

But when these processors are used for computing algorithms which are not highly regular, their performance degrades; moreover, they are also over burdened with complex programming problems; it is not possible for the programmer to extract all the parallelism in a problem.

A variety of topologies and designs for processor arrays exist and many com-

putational algorithms for these arrays have been proposed. These include globally synchronous systolic arrays and globally asynchronous wavefront arrays. A systolic array is a network of processors that rhythmically compute and pass data through the system. It is often algorithm oriented and is used as a co-processor with a host processor [1].

Information in a systolic array flows between processing elements (PEs) in a pipeline fashion, and the communication with the outside world occurs only at the boundary PEs. All operations, including the data movements in a systolic array, are controlled by a global clock (the design of which creates problems for very large arrays). On each global pulse, operands that arrive at any PE are read from its inputs and the corresponding results are produced simultaneously. Therefore, in a systolic array all motions occur synchronously with the global clock [1] and hence the clock period determines the array processing speed. But this synchronization is achieved at the expense of adjusting the global clock in accordance with the slowest local operation and the longest delays caused by any clock skew [2].

Wavefront arrays are asynchronous arrays that substitute the requirements of correct timing with that of correct sequencing. Logically, the computational front in these arrays advances exactly as it does in systolic arrays, the difference being that the speed of propagation is not necessarily uniform across the front. Both systolic arrays and wavefront arrays have a computation front that propagates according to a predetermined fixed sequence [3]. Consequently, these arrays prove to be very effective for executing highly regular algorithms. Many computationally demanding

problems do not exhibit high regularity and therefore are not suitable for these arrays. Therefore, in order to execute any arbitrary algorithm, a processor array must be capable of doing computations required by the algorithm even though there are no computational fronts.

The dataflow model of execution offers attractive properties for parallel processing. Firstly, it is asynchronous because it bases instruction execution on operand availability, synchronization of parallel activities is implicit in the dataflow model. Secondly, it is self-scheduling except for data dependencies in the program, dataflow instructions do not constrain sequencing; hence the dataflow graph representation of a program exposes all form of parallelism, eliminating the need to explicitly manage parallel execution.

Due to its simplicity and elegance in describing parallelism and data dependencies, the dataflow execution model has been the subject of many research efforts. The pioneering work in the field of dataflow processors was done by Jack Dennis [4] in the early 70's. His work forms the basis of the static dataflow architecture. The MIT dataflow architecture [5] is the direct implementation of the static dataflow model developed by Dennis. It consist of a set of processing elements interconnected through a communication network as shown in Figure 1.1.

A small engineering model consisting of four micro-coded processors was built [6], however it suffered from a number of drawbacks such as lack of support for procedure calls, non-strict operators and general recursion. Despite its drawbacks, the MIT static dataflow architecture has introduced new ways of thinking in the

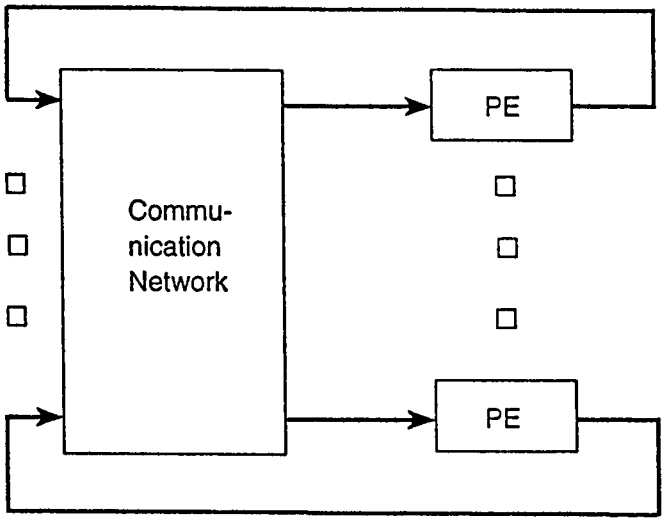

Figure 1.1: Communication structure of the MIT static dataflow architecture.

direction of massively parallel computers.

The MIT tagged token dataflow machine [5] was the first developed on the principle of tagged token, i.e., **a node is enabled as soon as tokens with identical tags are present at each of its input arcs.**

This machine consist of a number of PEs connected through an n-cube packet switching communication network to a set of specialized storage elements. It has been simulated on Multiprocessor Emulation Facility consisting of 32 TI Explorer Lisp Machines interconnected by a high speed network, which has been operational since 1986.

The experience gained with the MIT tagged token dataflow computer is being used to build a prototype of its successor, the Monsoon Architecture [7]. Currently a 64-bit single processor wire-wrap prototype of the Monsoon architecture is operational.

The same principle of tagged token dataflow was also developed independently at the university of Manchester. The resulting Manchester dataflow computer [8] is the first actual hardware implementation of a dynamic dataflow computer ever built. The inadequacy of handling complex data has been perhaps the most serious drawbacks of the Manchester dataflow computer. Though it was too small to run any actual applications, it was able to demonstrate that pipelines in a dataflow computer can be kept busy almost effortlessly.

The SIGMA-1 [9] is the most ambitious tagged token dataflow computer built to date. It is a supercomputer for large scale numerical computations and has been

operational since early 1988 at the Electro-Technical Laboratory in Japan. It consist of 128 PEs and 128 structure elements interconnected by 32 local networks and one global two-stage omega network. Sixteen maintenance processors are also connected with the structure elements and with a host computer for I/O operations, system monitoring, and maintenance operations.

The tagged token architecture has also suffered from certain drawbacks such as large amount of memory needed to store tokens which makes it impractical and inefficient. Moreover, they perform quite poorly with sequential code. This is due to the overhead associated with token matching, communication, instruction scheduling, and structure storage access.

All the above stated dataflow architectures were designed for very complex floating point operations with complex multichip hardware designs. A low-hardware-complexity dataflow PE for 8-bit fixed point operations was proposed by Silberman and his group [3]. This low complexity should make possible the fabrication of a VLSI chip containing about 50 to 100 cells. Several such cells can then be put together to form a larger processor [10]. This design has also suffered from serious flaws which were revealed during this research work.

In order to execute arbitrary algorithms effectively, a highly reconfigurable dataflow array processor has been designed in this research. To further increase the speed and reduce area of a VLSI chip, a RISC (reduced instruction set computer) architecture [11] has been used.

This thesis is organized as follows. In Chapter 2 basic concepts are discussed. In

this chapter a brief definition and comparison of dataflow and control-flow computing (Section 2.1, 2.2 respectively) is followed by a discussion on advantages of dataflow computing (Section 2.3). Execution models of dataflow computing are described in Section 2.4 and a dataflow graph with an example is illustrated in Section 2.5. Finally the peculiarities of a RISC architecture are given in Section 2.6.

Chapter 3 contains the design goals, the instruction set, and a complete design of the processing element (PE). The PE architecture design includes the ALU unit (Section 3.4). The remaining sections deal with instruction memory and data memory organization and design (Section 3.5), parallel dataflow and instruction execution control unit working and design (Section 3.6), configuration registers operation and organization (Section 3.7), direct matching unit design and dynamic dataflow execution (Section 3.25), and hardwire for instructions, instruction decoder, and control unit (Section 3.9).

Chapter 4 covers the array topology and design tools, followed by an example with simulation results.

Chapter 2

Basic Concepts

Dataflow computers are completely different from control-flow computers usually referred to as von Neumann machines. The basic difference lies in the execution of instructions. In control-flow the program is executed under the 'control' of a program counter, while in dataflow an instruction is executed on the arrival of all its inputs, without the interaction of a program counter.

In the following sections these two concepts are compared. In order to contrast these two computer architectures, a simple program in machine code will be presented.

2.1 Control-Flow

In a control-flow computer explicit control actions cause the execution of instructions. Traditional control-flow machines have a single thread of control, which is

passed from one instruction to the other by the help of a program counter. Program control transfers are caused by Jump, Call, and Return instructions.

As shown in Figure 2.1, all the instructions for the execution of the expression, $(a+b) \times (c+d)$ are arranged in a sequential order and the program counter is used to point to a particular instruction. The correct sequencing of the instruction is very important, otherwise it would result in faulty computation.

All these instructions share the data, by placing data in memory cells. The data is accessed by all instructions using pointers embedded in the instruction itself. Each instruction consists of an operator, followed by one or more operands which are literal or numerical values. The instruction format is shown in Figure 2.2.

In Parallel Control-Flow computers, special parallel control operators like Fork and Join are used to specify parallelism explicitly. These operators allow more than one thread of control to be active at an instant and provide a means for synchronizing these threads as shown in Figure 2.3.

The main features of control-flow are summarized as follows:

- Program counters are used to control the execution of instruction in a centralized control environment.
- Procedure calling require explicit control operators.
- Data is passed between instructions via references to shared memory cells.
- Flow of control is implicitly sequential, but special control operators are used for parallelism.
- Flow of data and control are different.

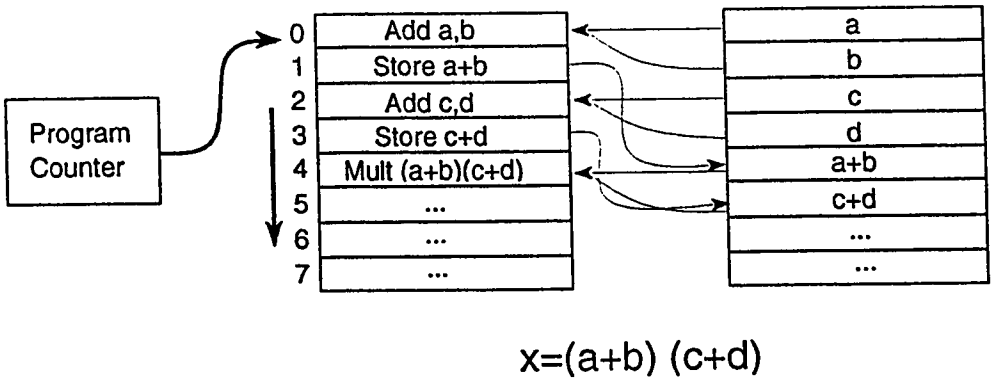

Figure 2.1: Program execution in a von Neumann computer.

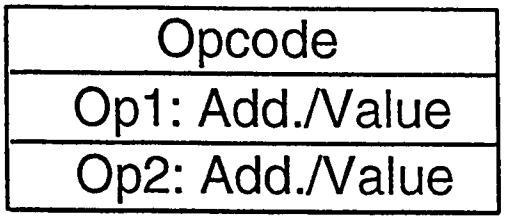

Figure 2.2: Instruction format in a von Neumann computer.

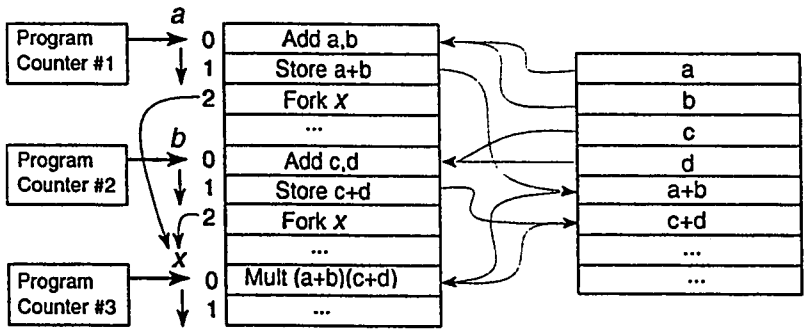

Figure 2.3: Program execution in a parallel control-flow computer.

2.2 Dataflow

The idea of dataflow computing is rather old, but its development is still in the infant stage [12]. Due to parallel computing problems with control-flow, and due to advancement in VLSI technologies, it has gained momentum. In a dataflow computer, an instruction is executed on the arrival of all its operands; that is when the instruction has received all its inputs. In dataflow, instructions share data via the producing instructions that pass a separate copy of the data (known as data tokens) to each consuming instruction [13].

Thus flow of data and flow of control are tied together by a single scheme, data tokens. As shown in Figure 2.4 black dots on arcs are used to represent the availability of data tokens. The nodes perform the required operation, and using the embedded reference, copy the result tokens into each consumer instruction.

Each instruction consists of an operator, one or two source operands, and one or more destination operands, to which the results (data tokens) will be sent, as shown in Figure 2.5. A data token is formed by combining the result value and its destination.

Many of these tokens or packets are passed among various resource sections of a dataflow machine. Therefore a machine can assume a packet communication architecture which is a type of Distributed Multiprocessor Organization [14]. The main features of dataflow are summarized as follows:

- Data is passed directly between instructions.

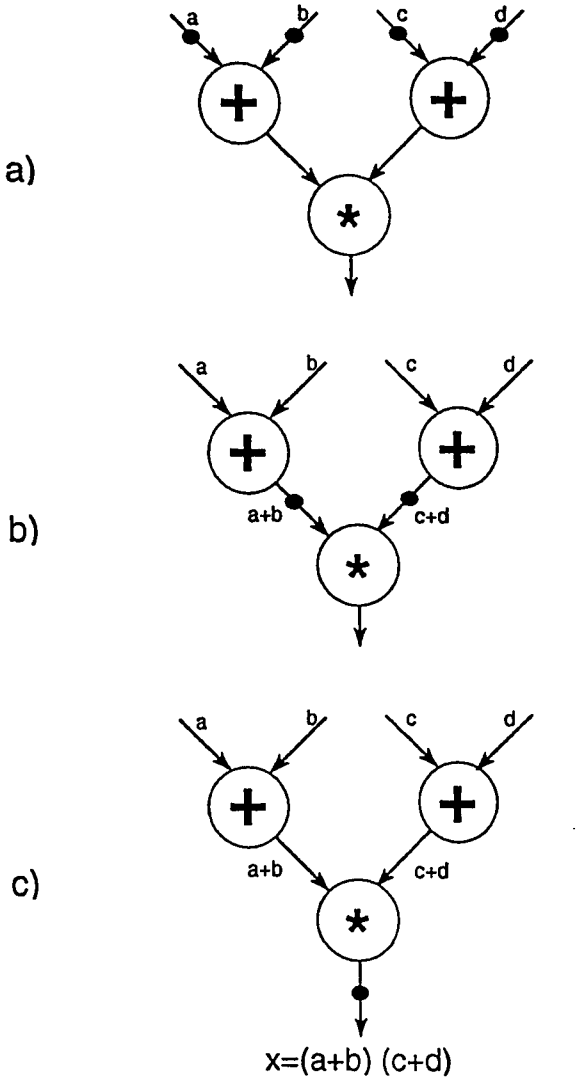

Figure 2.4: Program execution in a dataflow computer.

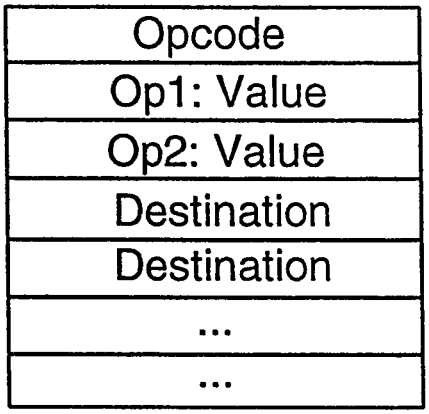

Figure 2.5: Instruction format in a dataflow computer.

- There is no concept of shared memory.
- Procedure calling requires explicit control operators.
- Program sequencing is constrained only by data dependencies among instructions.
- Flow of control is tied to flow of data.

2.3 Advantages of Dataflow Computing

Control-flow is synchronous, because the program execution is controlled by a centralized control unit. In control-flow parallel computers, free access of global variables by different procedures can produce undesired side effects. An unexpected order of execution or manipulation of global variables can cause errors in computational results. To achieve highly parallel processing, a programmer must exactly describe which instructions are to be executed concurrently, without causing side effects. Increase in the number of blocks (sets of instructions and functions) for parallel execution also increases the complexity of program structure. Finally the programmer faces the difficult task of trying to extract all the parallelism of the target problem [15].

Dataflow is asynchronous, which means that many instructions can be executed simultaneously and asynchronously, depending on the arrival of data tokens. It offers a possible solution to the problem of efficiently exploiting concurrency of

computation on a large scale. Thus highly concurrent computation is a natural consequence of the dataflow concept.

Direct use of values instead of addresses enables purely functional programming without side effects. Since there is no use of shared memory cells, dataflow is purely functional and free from side effects, such as changes of memory words. In a dataflow processor, operands are passed directly as data tokens instead of addresses of the operands. These features make dataflow computers suitable for parallel and distributed implementation. In theory it is possible to achieve maximum parallelism if sufficient resources are available [16].

2.4 Dataflow Execution Models

Depending on how a dataflow computer handles data tokens, two models for dataflow execution exist:

- Static dataflow.
- Dynamic dataflow.

In the following paragraphs these two execution models are described and compared.

2.4.1 Static dataflow model

This dataflow model enables a node as soon as

- all its tokens are present on its inputs, and,

- no token is present on any of its output arc.

In this scheme control tokens are used to acknowledge the transfer of data tokens among nodes. For example, consider Figure 2.6. In part a, two operands $a1$ and $b1$ are present at the input of node $n1$, and operands $a2$, $b2$ are still in the queue. Since no token is present at the output of node $n1$, node $n1$ is fired to produce $c1 = a1 \times b1$. In part b, $a1$, $b1$ are removed from the input of node $n1$ and $a2$, $b2$ are applied at the input of node $n1$. But node $n1$ will not fire until it gets an acknowledge signal from *node* $n2$. Once node $n1$ gets an acknowledge signal from the node $n2$ it produces the result $c2 = a2 \times b2$.

This dataflow model was proposed by J. B. Dennis [12]. In this dataflow model each instruction template contains the opcode, a slot for operands and the destination addresses.

2.4.2 Dynamic dataflow model

In this dataflow model a node is enabled as soon as

- tokens with identical tags are present at each of its input arcs.

Instead of acknowledge signals, tags are attached to data tokens in this model. These tags uniquely identify the context of that particular token. This technique is also called tagged token, because the tagging is achieved by attaching a label with each token. These tags are matched by a matching unit before instruction execution.

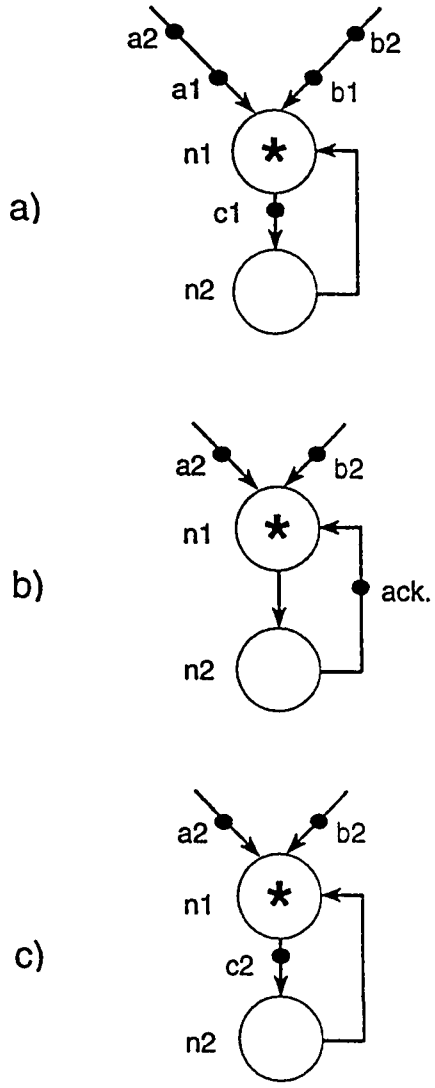

Figure 2.6: Static dataflow model.

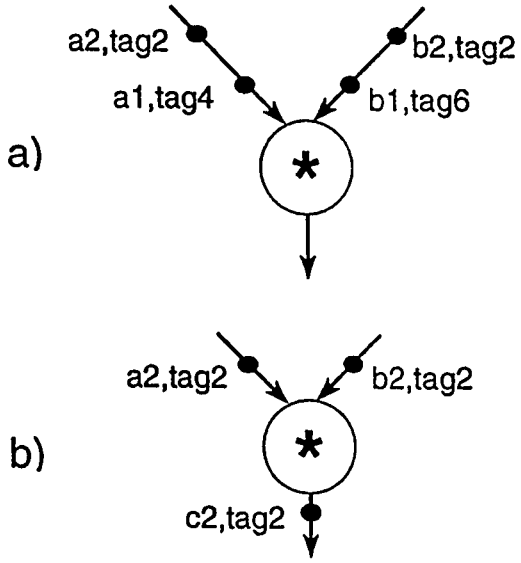

Figure 2.7: Dynamic dataflow model.

As shown in Figure 2.7, two tokens $a1$, $b1$ with different tags are present at the input of the node. Since the tags are different the tokens are discarded by the matching unit of the node and no output is produced. But when tokens $a2$, $b2$ with similar tags $tag2$ are placed at the input of the node instruction execution takes place and a result token with a new tag is produced [17].

This model was proposed by Arvind at MIT [5] and Gurd, Watson [8] at the University of Manchester. In this dataflow model each instruction consist of an opcode, a literal/constant field, and several destination fields.

2.4.3 Static dataflow *vs* Dynamic dataflow

The static dataflow model is easy to implement. It is very easy to detect the data arrival by the help of a single (presence/newdata) bit, for instruction execution. However, it has a performance drawback when dealing with iterative constructs and reentrancy [16]. The acknowledge scheme can transform a reentrant code into an equivalent graph that allows pipelined execution of consecutive iterations, but this transformation increases the number of arcs and tokens. More important, it exploits only limited amount of parallelism, since the execution of consecutive iterations cannot fully overlap, even if no dependencies exist. Although this inefficiency can be alleviated in case of loops by providing multiple copies of the program graph, the static dataflow model lacks the general support for programing constructs essential for any modern programing environment.

The major advantage of the dynamic dataflow model is the higher performance

it obtains by allowing multiple tokens on an arc. Despite the potential of dynamic dataflow model for large-scale parallel computer systems, previous experiences have identified a number of shortcomings:

- overhead involved in matching tokens is heavy,
- resource allocation is a complex process,
- the dataflow instruction cycle is inefficient, and
- handling data structures is not trivial.

Detecting matching tokens is one of the major aspects of the dynamic dataflow computation model.

2.5 Dataflow Graph

The dataflow approach represents an algorithm (arithmetic expression) in the form of Dataflow Graph, and then maps it on the processor array.

A dataflow graph is a directed graph of labeled nodes, representing instructions, and of arcs, which represent data dependencies among nodes. Operands are propagated along the arcs in the form of data packets called tokens. A dataflow graph shows the parallelism embedded within the target problem and describes it naturally and this is one of the most promising programing schemes for parallel computing. A programmer can indirectly define the existence of parallelism between blocks by describing their data dependencies.

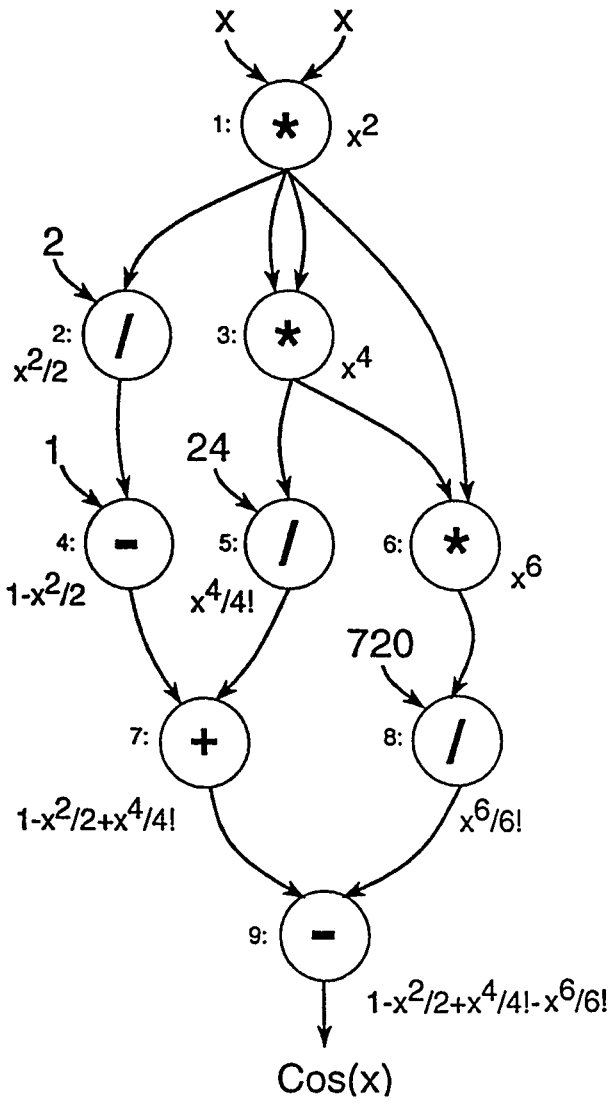

Figure 2.8: Dataflow graph of $Cos(x)$.

For example a *Cosine* function can be approximated by the following expression:

$$Cos(x) = 1 - x^2/2! + x^4/4! - x^6/6!$$

This expression can be represented in the form of a dataflow graph as shown in Figure 2.8. In this figure, operations such as addition or multiplication are represented by nodes (circles) which are interconnected by directed arcs (arrows). These arcs represent the data dependencies among the nodes and the path of dataflow.

This scheme ensures that parallelism is inherent in the problem. Therefore, the efficiency of the resultant program is virtually independent of the programmer's ability to find possible parallelism in the target problem. In the above example, the parallelism is obvious, x^2 is first generated and then x^4, x^6 and $x^2/2$ are calculated using x^2.

The dataflow graph can easily be subdivided by breaking the arcs that connect the modules. This helps to simplify mapping the whole program into various multiprocessors.

2.6 RISC Architecture

A Reduced Instruction Set Computer (RISC) Architecture has the following key features [18] [19]:

- Single cycle execution.
- Simple load/store design.

- Hardwired control.
- Simple fixed format and fixed length instructions.
- Simple addressing modes.
- Register based execution.
- Large number of registers.

The single cycle operation in a RISC architecture facilitates the rapid execution of simple functions that dominate a computer's instruction stream and promotes a low interpretive overhead. Load/Store design follows from a desire for single-cycle operation.

Hardwired control provides for the fastest possible single-cycle operation. Microcode implemented control leads to slower control paths and adds to interpretive overhead. Moreover, microcode requires the maximum amount of area in the processor while the hardwired control requires the minimum area. The saving in area is used to introduce large number of registers in RISC architecture, which further increases the performance of the processor.

Relative small sets of instructions and addressing modes facilitate a fast, simple interpretation by the control engine unit. Fixed instruction format with consistent use eases the hardwired decoding of instructions which again speeds control paths.

Chapter 3

Processing Element Design

3.1 Design Goals

Many simulation studies of dataflow processor systems have been reported but only a few dataflow processors have been constructed. The ones that have been constructed such as Sigma-1, Epsilon-2 are very complex, and it is not easy to implement them on a single VLSI chip. In order to join the trend to dataflow computing a low complexity dataflow array processor with the following aims has been designed.

1. The dataflow processor should be highly reconfigurable, so that it
   - executes arbitrary algorithms,
   - is fault tolerant, and,
   - gives maximum flexibility to the programmer.

2. The processor should possess all the features of a static and dynamic dataflow models.

3. Since high performance is required, RISC architecture should be used.

4. The hardware should be simple, so that the processor could easily be implemented on VLSI.

5. The architecture should be easily extendible to massively parallel supercomputers of the future.

Architectural reconfiguration permits the realization of performance improvements, possible in an adaptable computer system, since the system can reconfigure itself into a sequence of best architectural states that take into account the computational specificities of each program. In adaptable computers each program is preprocessed to find the concrete characteristics of the best architecture for its execution. Once high performance is achieved the program is mapped on the adaptable computer system. The reconfiguration also gives maximum flexibility to the programmer to program arbitrary algorithms in the manner he likes, depending on his experience and heuristic. With the help of reconfiguration, the programmer can switch the system architecture into those states that match the peculiarities of computed algorithms.

In current VLSI technology, a significant number of production defects occur in the process of manufacturing VLSI chips, which reduces yield significantly. Since a processor array often requires real-time, maintenance-free, uninterrupted operation

for long periods of time, especially in harsh environments, there must be a systematic procedure for reconfiguring processor array in the presence of faulty processors. This reconfiguration capability guarantees fault tolerance in applications, where a failure is not permissible.

Although much work has been done in the reconfiguration of processor arrays in the presence of faulty components, there has been no work done concerning reconfiguration of a processor array that executes arbitrary algorithms in the framework of multidimensional reconfiguration and graceful degradation.

Static dataflow is easy to implement and is best suited for many numerical applications that demand the computing power of a supercomputer [17]

The tagged token architecture appears best suited for applications such as symbol manipulations, iterative constructs and reentrancy that have a less regular structure. It can efficiently control the parallel and concurrent execution of multiple functions.

The static dataflow model has performance degradation when dealing with iterative constructs and reentrancy, and tagged token dataflow performs quite poorly with sequential code. This is due to the overhead needed with token matching, communication, instruction scheduling and structure storage access, which cannot be masked by the pipelining effect of multiple execution threads.

Therefore the simultaneous use of dynamic and static execution models has the advantage of avoiding tagging, i.e., tokens which are not needed by several instructions do not need tagging, or dynamic execution. This not only increases the processor performance but also makes the matching unit design easy.

As has been already mentioned earlier, most of the dataflow computers realized are too complex to be implemented on single VLSI chip. For example, the board size of Epsilon-1 was 1 × 1 foot and required 60 watts. Hence it was decided to keep the design as simple as possible to accomplish VLSI implementation. RISC architecture is the best choice for a simple design. It uses a small set of simple instructions and a simple control unit with high clock rates. The RISC approach not only requires less area but also gives high performance in terms of speed. It is not possible to design a high performance parallel computer without a RISC architecture. In dataflow computing a RISC architecture is not welcomed due to overhead needed for token matching in complex operations. But because of the static RISC architecture this processor is able to implement difficult instructions and routines without performance degradation due to token matching.

3.2 Instruction Set

One of the goals of designing a dataflow RISC array (DF-RISC-A) was to obtain high performance with as little complexity as possible. Therefore, before defining the instruction set, the complexity and performance of each instruction was carefully evaluated using VHDL simulations and heuristics. After a thorough scrutiny of 32 instructions 25 were selected for this processor array. All these instructions are 16-bit long, having similar format and single register to register addressing mode.

A two operand instruction (see Figure 3.1) consists of an $Op-code$ field (bits

$13-9$), Destination register address (bits $8-6$) and *Operand*1 & *Operand*0 register addresses (bits $5-3$ and $2-0$ respectively). The single operand instructions have the same format, except that the *Operand*1 field is empty. This format requires a simple instruction decoder, an added advantage of using a RISC architecture.

This format has been chosen after simulating different instruction formats on VHDL. One format suggested by Koren [10, 3] uses undecoded bits for operand selection. This approach seems to be very attractive, because it does not require decoding. But it is very difficult to implement, moreover, it is very difficult to differentiate between first and second operand, and a maximum of two operands can be specified, not including the destination operand. This approach also increases the instruction length. These facts were revealed after VHDL simulations.

All instructions are executed in four phases:

1. Select and decode instruction.
2. Fetch operands from registers.
3. Execute instruction.
4. Write results.

The instructions are grouped into following four categories:

- Arithmetic,
- Logical,

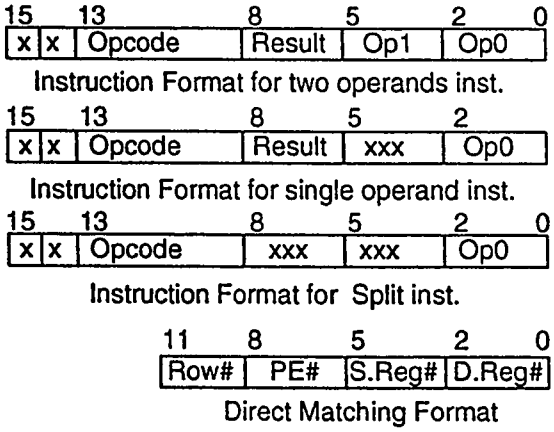

Figure 3.1: Instruction Format of DF-RISC-A.

- Move and Flow Control, and

- Split and Join Instructions.

Dataflow instructions are not executed in sequence, hence a *carry* generated during arithmetic operations must be saved. It will be stored in the R_d+1 register that is, the register next to the destination register R_d. It is up to the programmer to use this carry or not. This technique helps in high precision arithmetic operations.

In unsigned multiplication the number of bits of the *product* is equal to the sum of bits of *multiplicand* and *multiplier*. For 8 × 8 multiplication a result of 16 bits is produced. In this dataflow processor, the least significant byte is stored in the destination register R_d and the most significant byte is stored in the next register to the destination register, that is R_d+1. The same rule applies for a *quotient* and *remainder* produced during division operation, i.e., the quotient is stored in the destination register R_d and the remainder is stored in R_d+1.

In order to execute single cycle division and multiplication, a special combinational hardware multiplier [20] and divider has been designed with low *area* × *time* factor, instead of software routines as normally done in RISC processors.

The dataflow control instructions are used to control the program flow based on different conditions. These instructions are used with the *compare* instruction and transfer data depending on the flag setting. The *split* and *join* instructions are best explained after having discussed the architecture design.

Table 3.1: Table illustrating the RISC instruction set.

Instr.	Opcode	Operands	Comments	
Add	00001	Rs_1, Rs_2, Rd	$Rd \leftarrow Rs_1 + Rs_2$	integer add
Sub	00010	Rs_1, Rs_2, Rd	$Rd \leftarrow Rs_1 - Rs_2$	integer subtract
Mul	00011	Rs_1, Rs_2, Rd	$Rd \leftarrow Rs_1 \times Rs_2$	integer multiply
Div	00100	Rs_1, Rs_2, Rd	$Rd \leftarrow Rs_1 \div Rs_2$	integer divide
Cmp	01001	Rs_1, Rs_2	$Rs_1 - Rs_2$	1's compare
Inc	01010	Rs, Rd	$Rd \leftarrow Rs + 1$	increment
Dec	01011	Rs, Rd	$Rd \leftarrow Rs - 1$	decrement
Neg	01011	Rs, Rd	$Rd \leftarrow \bar{R}s - 1$	2's compl.
AND	00101	Rs_1, Rs_2, Rd	$Rd \leftarrow Rs_1 \& Rs_2$	logical AND
OR	00110	Rs_1, Rs_2, Rd	$Rd \leftarrow Rs_1 \bullet Rs_2$	logical OR
EXCL	00111	Rs_1, Rs_2, Rd	$Rd \leftarrow Rs_1 \oplus Rs_2$	logical EXOR
Not	01101	Rs, Rd	$Rd \leftarrow \bar{R}s$	1'compl
Shl	01110	Rs, Rd	$Rd \leftarrow Rs \vert 0$	shift left
Shr	01110	Rs, Rd	$Rd \leftarrow 0 \vert Rs$	shift right
Mov	10000	Rs, Rd	$Rd \leftarrow Rs$	move
Movz	10001	Rs, Rd	$Rd \leftarrow Rs$	move if zero
Move	10010	Rs, Rd	$Rd \leftarrow Rs$	move if equal
Movl	10011	Rs, Rd	$Rd \leftarrow Rs$	move if less
Movg	10010	Rs, Rd	$Rd \leftarrow Rs$	move if greater
Creg	11000	Rd	$Rd \leftarrow 00$	clear register
Clrc	11001	Rd	$Rd \leftarrow 00$	clear carry bit
Setc	11010	Rd	$Rd \leftarrow 01$	set carry bit
Splt	10101	Rs	$BUS \leftarrow Rs$	broadcast Rs
Spltg	10110	Rs	$G - BUS \leftarrow Rs$	glob. broadc. Rs

$\star$ note: There is no instruction for the Join statement. which is implemented by means of DIRECT MATCHING TECHNIQUE.

3.3 Architecture Design

Based on the design goals of Section 3.1 and the instruction set shown in Table 3.1, a processing element architecture has been designed (see Figure 3.2).

This processing element operates on 8-bit operands. All data registers are 8 bits wide. Two or more instructions must be used for the execution of 16-bit and 32-bit arithmetic operands. All processing elements are designed, with the same identification (ID) number. The processing element ID number is allotted during the initialization phase by the help of software. Each processing element can be subdivided (see Figure 3.2) into the following functional blocks:

1. Arithmetic Logic Unit (ALU).
2. Instruction memory and Data memory.
3. Parallel dataflow and instruction execution control unit.
4. Configuration Registers.
5. Direct matching unit.
6. Hardwired control and instruction decoder.

In the following pages the design and implementation of all these units is discussed in detail.

3.4 ALU Design

The various circuits used by the processor to execute the instructions are combined in a single unit called **Arithmetic-Logic-Unit** (ALU), which executes almost all

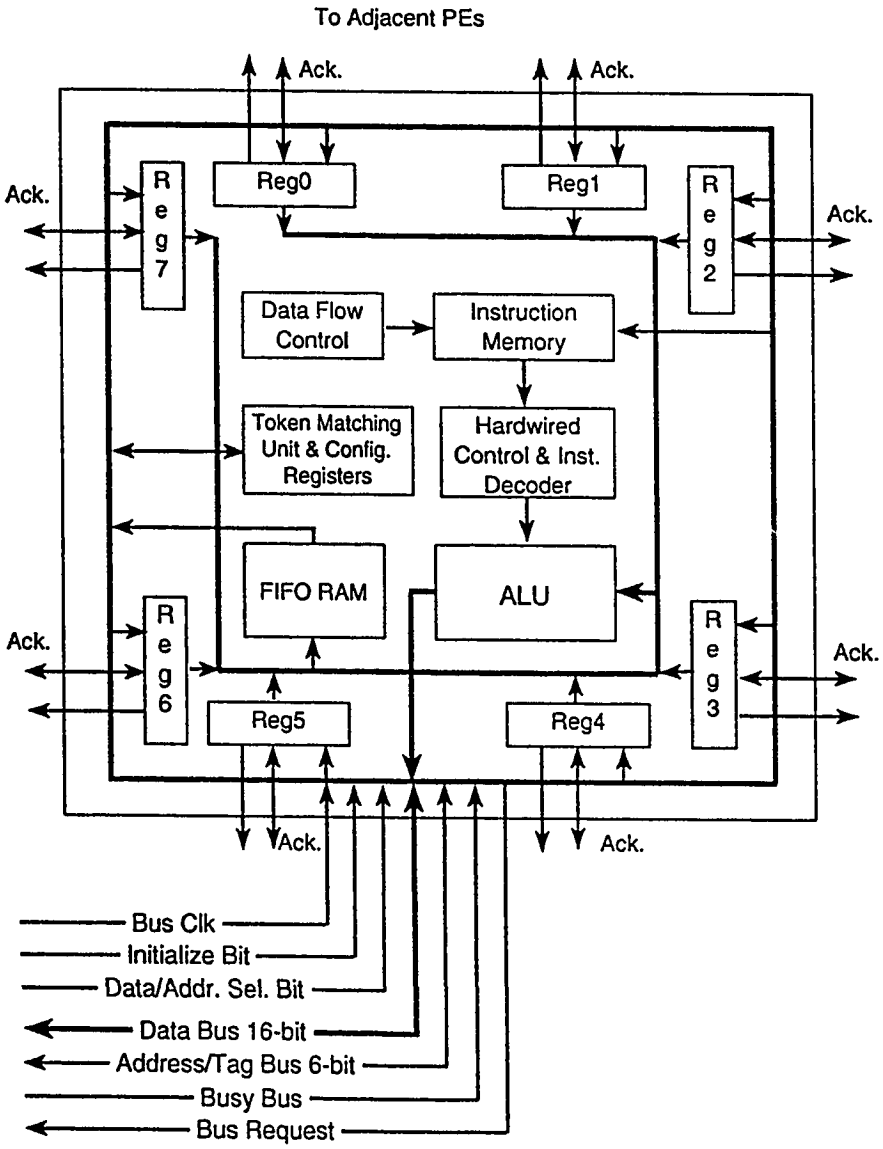

Figure 3.2: Block diagram of a processing element.

the instructions given in Table 3.1 on 8-bit operands. This ALU performs the arithmetic Add, Subtract, Multiply, etc., and the logical operations such as AND, OR, EXOR, etc.

Figure 3.3 shows the block diagram of the ALU. All the above stated operations can be performed by one or more of the blocks shown. For example the $shift-left$ operation is performed by the hardware shifters of the PP-cells of the multiplier and comparison is performed by the subtraction of two operands in the Add/Subtract Unit. In the following paragraphs first the design and implementation of the main blocks are discussed, and, then the complete ALU operation is explained by combining all these blocks.

3.4.1 Add/Subtract Unit

The Add/Subtract Unit is composed of a Carry Lookahead Adder and few EXOR gates. The carry lookahead adder (CLA), is the biggest block of the ALU. It is used by most of the operations such as addition, subtraction, multiplication, etc.

The general strategy in designing fast adders is to reduce the time associated with carry propagation. In Carry Lookahead Adders (CLA), the ripple carry effect is avoided by generating the input carry bit for each sum bit in an adder stage concurrently by using additional logic circuitry [21] [22]. An n-bit CLA can be formed from n stages, each of which is a full adder modified by replacing its carry output line C_i by two carry generate and propagate signals G_i and P_i defined by the logic equations

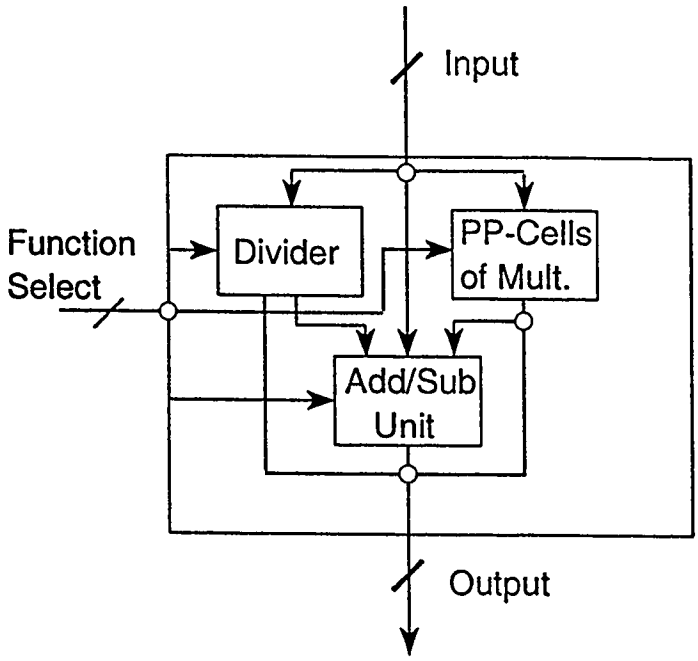

Figure 3.3: Block diagram of the ALU.

$$G_i = X_i \cdot Y_i \tag{3.1}$$

$$P_i = X_i \oplus Y_i \tag{3.2}$$

where, X_i and Y_i are the i^{th} bits of Operands X and Y.

The carry generate function G_i reflects the condition that a carry is originated at the i^{th} stage. The function P_i called carry-propagate, is true when the i^{th} stage will pass the carry C_i to the next higher stage. By substituting the expressions for P_i and G_i in the general equation of a full adder we get

$$S_i = (X_i \oplus Y_i) \oplus C_{i-1} = P_i \oplus C_{i-1} \tag{3.3}$$

$$C_i = X_i \cdot Y_i + (X_i \oplus Y_i) \cdot C_{i-1} = G_i + P_i \cdot C_{i-1} \tag{3.4}$$

where, S_i is the sum of the i^{th} bits of operands X and Y, C_{i-1} is the carry in and C_i is the carry out generated by the addition of i^{th} bits of the operands X, Y and C_{i-1}.

These equations reveal the fact that all P_i and G_i for $i = n-1, ..., 1, 0$ can be generated simultaneously from the external inputs X and Y. Equation 3.3 implies that the sum bits S_i for $i = n-1, ..., 1, 0$ can be generated in parallel, if all the carry inputs $C_{n-2}, ..., C_0, C_{-1}$ are available simultaneously. By repeatedly applying the recursive formula in Equation 3.4 a set of carry equations are obtained in terms of the variables P_i, G_i and the initial carry C_{-1}.

$$
\begin{aligned}
C_0 &= G_0 + C_{-1}P_0 \\
C_1 &= G_1 + G_0P_1 + C_{-1}P_0P_1 \\
&\vdots \\
&\vdots \\
C_{n-1} &= G_{n-1} + G_{n-2}P_{n-1} + ... + C_{-1}P_0P_1...P_{n-1}
\end{aligned}
\tag{3.5}
$$

These equations can easily be realized with a combinational logic circuit, known as an n-bit carry lookahead unit. Theoretically, one should be able to build a full CLA adder for any word length if the CLA unit can be freely expanded. Due to the fan-in and fan-out limitations of current CMOS gates, at present, single level CLA applies only to designing parallel adders of length $5 \leq n \leq 16$. CLA does not pay off for adders of length $n \leq 4$, because short ripple carry would be equally fast. For adders of length $n \geq 16$, multilevel carry lookahead is applied.

In this processor a CLA of 16 bits is needed, due to the 16-bit product. This CLA is used to generate a 16-bit sum of the *Sum* and *Carry* vectors produced by the carry save adders of the Partial Products cells (PP-cells of the multiplier). In add, subtract, increment, decrement, and compare operations, only 8 bits of the CLA are used. For subtraction and comparison, the subtrahend input of the CLA is inverted and added with *carry* to the minuend to get 2's complement subtraction. In addition, subtraction and comparison operations, bit 9 is used as a carry out. This CLA has been implemented in VLSI using the built-in CLA module available in the library of OASIS Logic3 [23]. The OASIS library uses a 4-bit block look-ahead

method to produce a fast logic for large input sizes without an excessive amount of circuitry.

3.4.2 Multiplier

This PE requires high speed multiplication to implement single cycle execution of each instruction. A special multiplier has been designed [20] to accomplish single cycle execution. It is based on a new algorithm. The multiplier has been fabricated and successfully tested. The design is based on non-overlapped scanning of 3-bit fields of the multiplier. The partial products of the multiplicand and three bits of the multiplier are pre-calculated using only hardwired shifts. They are then added using a tree of carry-save-adders, and finally, the sum and carry vectors are added using a carry-look-ahead adder. The high speed feature of this algorithm is due to:

1. Pre-calculation of partial products of non-overlapped 3-bit fields by hardwired shifting.

2. Addition of partial products using a carry-save-adder (CSA) tree.

3. A single addition of the carry and sum vectors using a carry-lookahead adder (CLA).

Basic Design

The basic idea behind the algorithm consists of first finding the *partial products* of the multiplicand (B) and 3-bits of the multiplier (A). The product of the multi-

plicand and a three-bit number (0 to 7) is obtained by means of *shift* and *add* operations. As shown in Table 3.2, multiplication by a three bit number can be expressed as a single addition of two multiplicands shifted by a fixed amount. For example, multiplication by six is expressed as the addition of multiplication by 4 and multiplication by 2 (multiplication by powers of 2 is accomplished by shifts only). The multiplication by 7 (111) requires three additions. Therefore, this operation is replaced by subtracting the multiplicand from 8 times the multiplicand. Subtraction is done by adding the two's complement of the subtrahend. In order to avoid the delay produced to obtain the 2's complement of multiplicand, the LSB of the three times shifted multiplicand is pre-set to one. Then, only the inverse of the multiplicand is added to it. That is,

$$A \times 111 = A \times 1000 + \overline{A} + 1 = A000 + \overline{A} + 1 = A001 + \overline{A}$$

where A000 represents three zeros catenated to A, or A multiplied by $2^3 = 8$.

The two partial products to be added in Table 3.2 can be considered as two operands of an adder, denoted by left-operand and right-operand. Note that the left-operand is multiplied by either 2, 4, or 8, whereas the right-operand needs to be multiplied by 1, 2, or $\overline{1}$, the last representing negation. Therefore, as shown in Figure 3.4, the two columns consist of hard-wired shifters which are enabled by a simple encoder logic. Details of the encoder logic are given in Figure 3.5. The logic design of the

Table 3.2: Multiplication by 0 through 7 expressed in terms of addition/subtraction of powers of 2.

A	$A \times B$
multiplier-field	Expressed as shift/add
0 0 0	$0 + 0$
0 0 1	$0 + 2^0(B)$
0 1 0	$0 + 2^1(B)$
1 0 0	$2^2(B) + 0$
0 1 1	$2^1(B) + 2^0(B)$
1 0 1	$2^2(B) + 2^0(B)$
1 1 0	$2^2(B) + 2^1(B)$
1 1 1	$2^3(B) - 2^0(B)$

encoder block that enables one of the three shifters in each block of the *pp-cell* of Figure 3.4 is given in Figure 3.5.

For a 9-bit wide multiplier-operand, three such *pp-cells* are required. Each *pp-cell* takes the entire multiplicand operand, and 3 bits of the multiplier operand. This is illustrated in Figure 3.6. In general, the number of *pp-cells* is equal to $p = \lceil \frac{n}{3} \rceil$, where n is equal to the number of bits of the multiplier operand A.

The outputs produced by these *pp-cells* are added using a three-level tree of CSAs to produce the sum and carry vectors. Note that by using the above *pp-cells*, the number of additions is considerably reduced. In this case the reduction is from 9 additions to 6. In general the decrease in the number of additions is from n to $\lceil \frac{2n}{3} \rceil$.

In the final stage, the outputs of these CSAs are added using a carry-lookahead-adder circuit. The output of this adder is the product of the 9-bit multiplier with the m-bit multiplicand (where m is the number of bits of the multiplicand). Note that the number of levels of circuitry remains the same for larger bits of the multiplicand, (except for the small increase in delay of the CLA). Increase in the number of bits of the multiplier causes little additional delay due to increase in the number of levels in the CSA tree and the increase in length of the CLA.

3.4.3 Design of 2's Complement Multiplier

For multiplication of signed 2's complement numbers, the design is similar, except that additional circuitry is included to accommodate correction. The multiplier

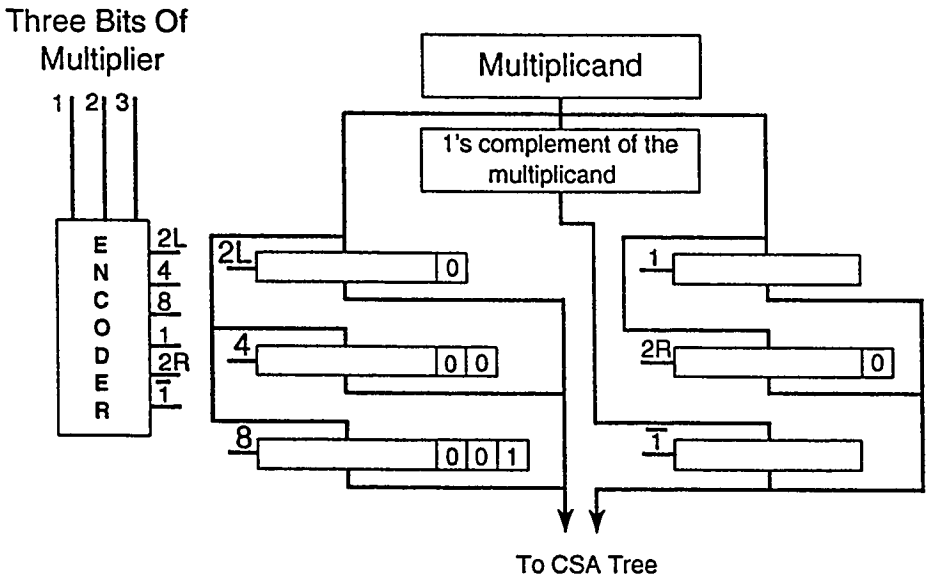

Figure 3.4: Block diagram of a *pp-cell*.

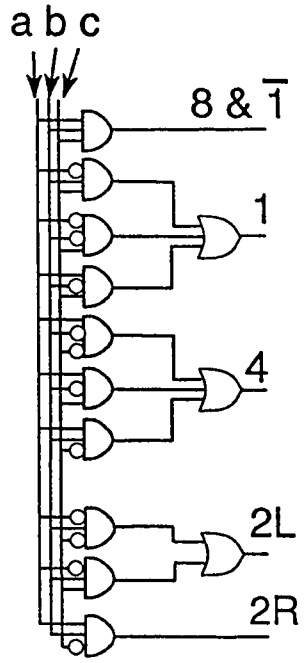

Figure 3.5: Logic diagram of the encoder cell shown in Figure 3.4.

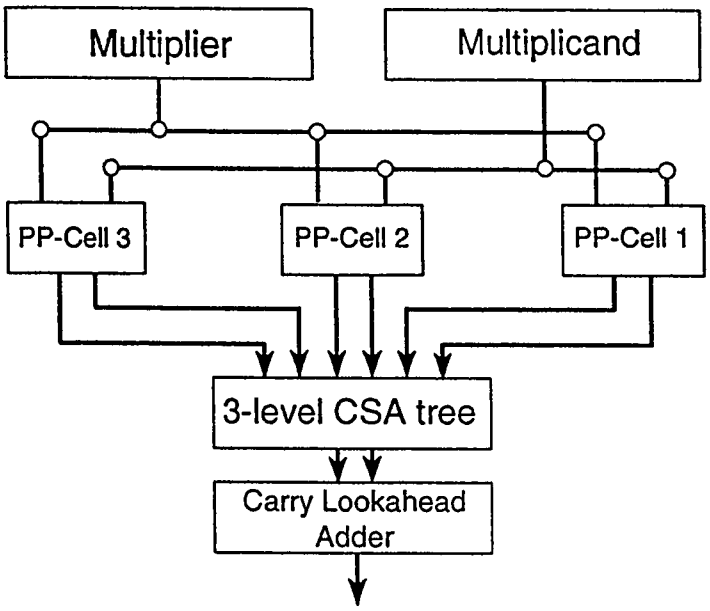

Figure 3.6: Block diagram of the 9-bit unsigned multiplier.

(A) is again divided into fields of at most 3-bits. The division of the multiplier into fields is slightly different from the case of the unsigned multiplication and is depicted below.

$A_{n-2}A_{n-3}A_{n-4}$	$\cdots$	$A_6A_5A_4$	$A_3A_2A_1$	A_0A_s

Observe that the sign bit ($A_s = A_{n-1}$) is grouped with the LSB of the multiplier. Bits A_1 to A_3 form one field, similarly bits A_4 to A_6 form the other, and so-on. The *pp-cell* required is identical to that in the previous multiplier for unsigned operands.

Correction Circuit

Generally, the 2's complement of an integer is obtained by complementing individual bits and then adding 1 to the number. One feature of the correction circuitry is that it avoids the delay due to rippling of carry produced by the addition of 1 in 2's complementation both before and after multiplication.

Two's complement numbers are accommodated by just inverting the negative operands (1's complement), and postcomplementation (1's complement) of the negative results. Two's complementation is avoided since it causes a delay due to addition of one. The error is adjusted using a correction factor as explained below. An additional circuit is used that generates in parallel a correction factor to be added to the result of 1's complement multiplication.

The technique that avoids the ripple delay due to adding a 1 is based on two

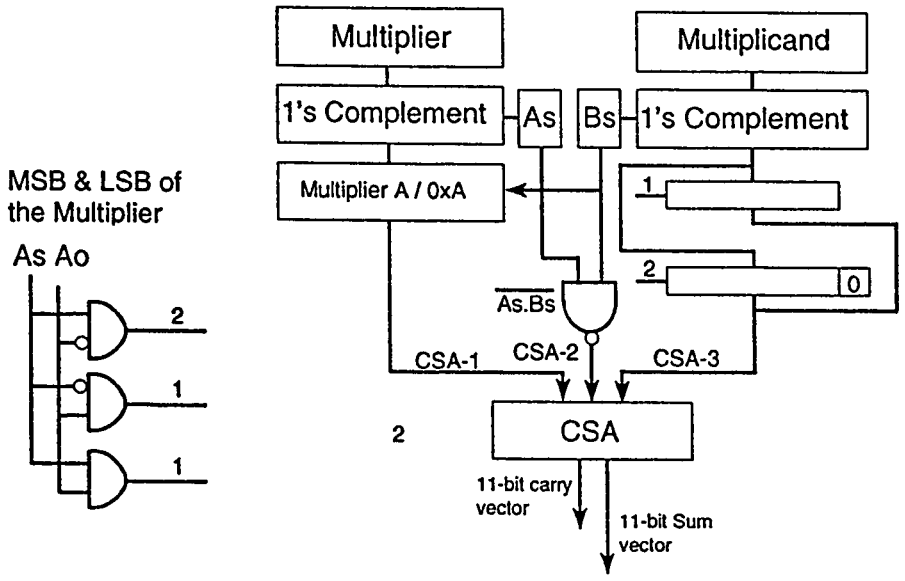

Figure 3.7: Correction circuitry for 2's complement multiplication.

observations (1) the distributive property of multiplication, that is, $A \times (\overline{B} + 1) = A \times (\overline{B}) + A$; therefore the addition of 1 is replaced by the addition of operand A to the result of 1's complement multiplication; and (2) the fact that the 2's complement of R is the same as the 1's complement of $(R - 1)$.

Based on the above two observations, an algorithm has been developed for generating a correction factor. The various values produced by the correction circuitry ($Corr$ or $Corr'$ if post complementation is required) are given in Figure 3.8, they have to be added to the output of the unsigned multiplier.

Below we explain one segment of the code when the multiplier (A) is negative (A_s=1) and the multiplicand (B) is positive (B_s=0).

When A_sB_s=10, the negative operand (A) is complemented and applied at the input of the encoder. The two operands are then multiplied using the method explained for unsigned numbers. The LSB (A_0) may be a zero or a 1. If A_0 is 1, then the complemented A_0 will produce a zero, and the correction factor due to adding a 1 (for 2's complement) will be the same as adding B later.

However, if A_0 was a zero, then the complemented A_0 would have produced a one, and the correction factor due to the addition of 1 (for 2's complement) will be the same as the addition of $2B$ later. In addition, since one of the operand is negative, it is required to find the 2's complement of the result later. This again is replaced by a simple inversion of the final result (1's complement) but subtracting a 1 in the correction circuit. Therefore, the correction circuitry output which must be

Case 0: $A_s = B_s = 0$

i) $A_0 = 0 \Rightarrow A$ is even

$P = AB$

$Corr = 0$

ii) $A_0 = 1 \Rightarrow A$ is odd

$P = (A-1)B + Corr = AB - B + Corr$

$P = AB$

$AB - B + Corr = AB$

$Corr = B$

Case 1: $A_s = 0, B_s = 1$

i) $A_0 = 0 \Rightarrow A$ is even

$P = A(B-1) + Corr$

$P = -AB = (\overline{AB-1})$

$AB - A + Corr' = AB - 1$

$Corr' = A - 1$ (Postcomplementation is required)

ii) $A_0 = 1 \Rightarrow A$ is odd

$P = (A-1)(B-1) + Corr = AB - B - A + 1 + Corr$

$P = -(AB) = \overline{AB-1}$

$AB - B - A + 1 + Corr' = AB - 1$

$Corr' = B + A - 2 = (B-1) + A - 1$

$Corr' = \overline{\bar{B}+1} + A - 1$ (Postcomplementation is required)

Case 2: $A_s = 1, B_s = 0$

i) $A_0 = 0 \Rightarrow \overline{\bar{A}+1} = A - 1$ is odd

$P = (\overline{\bar{A}+1} - 1)B + Corr = (A-2)B + Corr = AB - 2B + Corr$

$P = -(AB) = \overline{AB-1}$

$AB - 2B + Corr' = AB - 1$

$Corr' = 2B - 1$ (Postcomplementation is required)

ii) $A_0 = 1 \Rightarrow (\overline{\bar{A}+1}) = A - 1$ is even

$P = (\overline{\bar{A}+1})B + Corr = (A-1)B + Corr = AB - B + Corr$

$P = -(AB) = \overline{AB-1}$

$AB - B + Corr' = AB - 1$

$Corr' = B - 1$ (Postcomplementation is required)

Case 3: $A_s = B_s = 1$

i) $A_0 = 0 \Rightarrow (\overline{\bar{A}+1}) = A - 1$ is odd

$P = (\overline{\bar{A}+1} - 1)(\overline{\bar{B}+1}) + Corr = (A-2)(B-1) + Corr = AB - 2B - A + 2 + Corr$

P = AB

$AB - 2B - A + 2 + Corr = AB$

$Corr = 2(B-1) + A = 2(B-1) + (A-1) + 1 = 2(\overline{\bar{B}+1}) + (\overline{\bar{A}+1}) + 1$

ii) $A_0 = 1 \Rightarrow (\overline{\bar{A}+1}) = A - 1$ is even

$P = (\overline{\bar{A}+1})(\overline{\bar{B}+1}) + Corr = (A-1)(B-1) + Corr = AB - A - B + 1 + Corr$

P = AB

$AB - A - B + 1 + Corr = AB$

$Corr = A + B - 1 = (A-1) + (B-1) + 1 = (\overline{\bar{A}+1}) + (\overline{\bar{B}+1}) + 1$

Figure 3.8: The four cases of 2's complement multiplication.

Table 3.3: Table illustrating the third input of the CSA of correction circuit.

A_s	A_0	CSA-3
0	0	0
0	1	B
1	0	2B
1	1	B

added to the result is either $B-1$ (for $A_0 = 1$) or $2B-1$ (for $A_0 = 0$) (see Figure 3.8).

This correction factor ($Corr$) is added to the output produced by the other partial product cells using the carry-save adder tree. If one of the operands is negative, then the final output of the CLA is complemented.

The complete mathematical proof for the output produced by the correction circuitry is given in Figure 3.8 and summarized in Table 3.4. The mapping of this algorithm into hardware is shown in Figure 3.7 and yields in an extremely simple and fast correction circuitry. The correction circuitry output is produced in parallel with the output of the *pp-cells*, and is added to the sum of partial products using the carry-save adder tree.

Design of Correction circuit

The correction circuitry consists of a carry-save adder with inputs arriving from three sources. The three inputs of the CSA are labeled CSA-1, CSA-2 and CSA-3. The different values received by these inputs depend on the sign of the multiplier/multiplicand and the value of bit A_0. These values are summarized in Tables 3.3 and 3.4.

Table 3.4: Table illustrating the i_{th} input received by the CSA adder of the correction circuitry. *CSA-i* represents the i_{th} input. For input *CSA*-3 see Table 3.3.

As	*Bs*	$\overline{As \cdot Bs}$	A	B	*CSA-1*	*CSA-2*	$C_{in}CLA$	*Invert output.*
0	0	1	No change	No change	0	-1	1	0
0	1	1	No change	Inverted	A	-1	0	1
1	0	1	Inverted	No change	0	-1	0	1
1	1	0	Inverted	Inverted	A	0	1	0

3.4.4 Divider

In nearly all the computers, division takes the maximum amount of time. In this RISC dataflow array processor, special attention has been given to the design of a fast multiplier and divider, because multiplication and division are the most frequently used functions in digital signal processing, and parallel computing. If the multiplier and divider work at low speed then the single cycle execution cannot be achieved unless a slow clock is used, which results in performance degradation.

The multiplier design has been discussed in Section 3.4.2. In this section the design of an array divider will be given. The design is based on the non-restoring division algorithm. It is essentially the same array design proposed by others [24, 25, 26], but several design changes lead to a much faster divider.

In non-restoring binary division, successively right shifted versions of the divisor are subtracted from or added to the dividend and resulting partial remainders. The sign of the partial remainder determines the quotient bit and whether to add or subtract the shifted division in the next stage. The basic steps of the non-restoring algorithm can be stated with the help of the following notations.

Dividend: $n = n_{2j}, ...n_2, n_1, n_0$

Divisor: $d = d_j, ...d_2, d_1, d_0$

Partial Remainder: $r = r_j, ...r_2, r_1, r_0$

Final Remainder: $s = s_j, ...s_2, s_1, s_0$

Quotient: $q = q_j, ...q_2, q_1, q_0$

A $j+1$ bit quotient can be generated from an $j+1$-bit divisor and $2j$-bit dividend by the following non restoring binary division algorithm.

FOR $k = 0$ to j DO

*Step*1: Generate new partial remainder:

if $q_{j-1} = 1$

then
$r \leftarrow r - d$;
else
$r \leftarrow r + d$.

*Step*2: Quotient bit: $q_j = \overline{r}_j$

*Step*3: Shift partial remainder: $r \leftarrow 2r$

A two-dimensional array of iteratively structured logic can implement the non-restoring algorithm directly. Basically the array consists of rows of carry propagate adders with one control input per bit. Each element of the row consists of a full adder and an EXOR gate [24] as shown in Figure 3.9. The EXOR gate controls the divisor input to the full adder. The control signal S determines whether an add or subtract is to be performed. Subtraction is performed in 2's complement by forming the 1's complement of the divisor and forcing a carry into the low order bit position. The carry-out from from each row of the array is equal to the quotient bit for that row or bit of dividend. A non-restoring array divisor using ripple carry full adders requires execution time of $O(i^2)$ due to carry propagation.

requires execution time of $O(i^2)$ due to carry propagation.

The basic idea of this design is to eliminate the carry-ripple time, which is proportional to i, along each row. The best way to eliminate the carry propagation is to use a carry lookahead adder, instead of ripple carry adders. Subtraction of the divisor is again implemented using 2's complement addition as in the basic array. The design for the divider (15-bit dividend and 8-bit divisor) is shown in Figure 3.10.

The time taken by the carry lookahead adder (to generate the carry out) and EXOR gate (to invert) is the basic factor determining the delay per bit of the quotient q being produced. In the first stage inverters replace the EXOR gates (to reduce the time delay). Since the time delay for a CLA unit is 8δ for $i \leq 16$ and the time delay for EXOR gate is 3δ [21], the total delay for this divider is given by.

$$\Delta = 8\delta i + 3\delta(i-1)$$

$$= 11\delta i - 3\delta$$

The time delay required by different proposed non-restoring dividers are compared with this design in Table 3.5.

This design has been implemented using the CLA unit discussed in Section 3.4.1. In order to get the correct remainder in nonrestoring division, the last partial remainder produced during the non restoring division operation has to be restored if the last quotient bit is zero. The restoration is accomplished by adding the partial

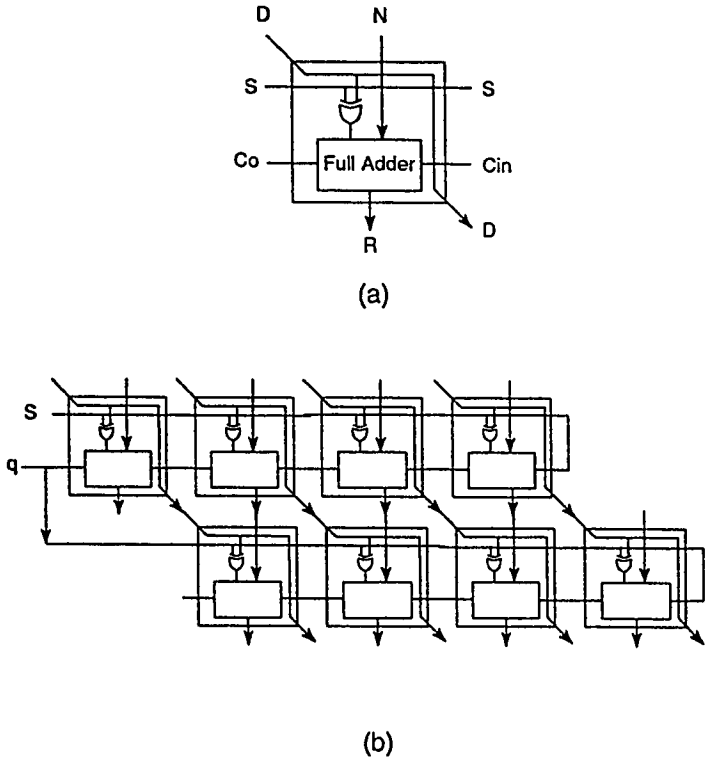

Figure 3.9: Non-restoring array divider (a) Basic cell. (b) Combinational Divider.

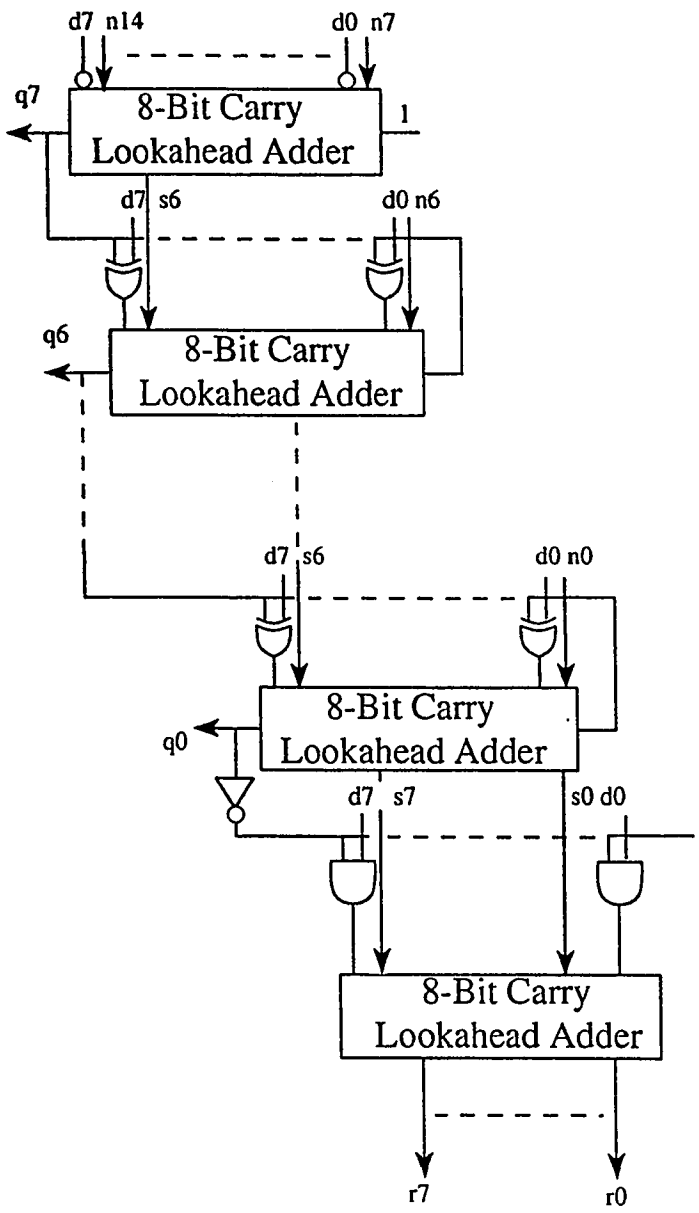

Figure 3.10: Proposed CLA divider.

Table 3.5: Comparison of Array Dividers.

TypeofDivider	*TimeDelay*	*No of bits i*
Ripple Carry Adders	$2i^2\delta + 7i\delta + 5$	
IAD (Cappa-Hammacher)	$12i\delta + 12\delta$	$i \leq 9$
IAD (Cappa-Hammacher)	$14i\delta + 12\delta$	$10 \leq i \leq 32$
Proposed Divider	$11i\delta - 3\delta$	$i \leq 15$
Proposed Divider	$13i\delta - 3\delta$	$16 \leq i \leq 64$

remainder with the divisor. In this divider the addition of the last remainder with the divisor is accomplished by using the 16-bit CLA of the multiplier, add-subtract unit. The VLSI implementation of this design uses the CLA, EXOR gates, and Inverters present in the library of OASIS Logic3.

3.4.5 ALU Operation

The ALU operation is simple and is explained with reference to Figure 3.11. The operation to be performed is selected by an encoder Figure 3.12. For a single operand instruction, only *Operand*1 and for a double operand instruction both operands are entered.

The 1-bit shift left operation is performed by the hardwired shifter of the multiplier, (the outputs of all the shifters are controlled by AND-OR logic); for shift right operation a new hardwired shifter has been added to the shifters of the multiplier. The output of these shifter is passed through the carry save adder tree then applied at the input of the CLA unit and finally sent to the output.

Addition, subtraction, increment, decrement operations are performed by the CLA of the multiplier unit. In these operations the second operand is selected from the hardwired shifters of the multiplier, using an encoder. For add, increment, and decrement, *Operand*1 without any shift is selected. However for subtraction by 2's complement addition, the inverted *Operand*1 is selected, the addition of one for 2's complement is performed by the CSA tree. The same operation is performed to execute the negate instruction, except there is no addition of *Operand*0. In

addition and subtraction *Operand*0 is applied directly at the input of the CLA using a multiplexer. The same multiplexer selects -1 (all ones) for decrement operation. Increment is performed by adding 1 through the carry in C_{in} of the CLA.

Comparison between two operands is performed by subtracting *Operand*1 from *Operand*0 and inspecting the *Zero* flag and carry out from the 8^{th} bit of the CLA, indicating different conditions outputs (> = <). Boolean operations are performed by using the AND, OR and EXOR gates. Inversion is again performed by selecting the inverted *Operand*1 from the multiplier hardwired shifters.

3.5 Instruction and Data Memory

In this processor array the instruction and data memory is distributed among all the processing elements. Each PE has its own instruction and data memory. We now discuss the organization and working of these memories.

3.5.1 Instruction Memory

Each processing element has its own instruction memory to hold eight instructions. Since the instruction length is 16 bits, the size of this memory is 8×16 bits. The instructions specify the operations each processing element has to perform on its operands. Therefore each PE can execute a small portion of a dataflow graph independently and concurrently with other processors. These instructions are loaded from the host computer during the initialization phase and remain unchanged

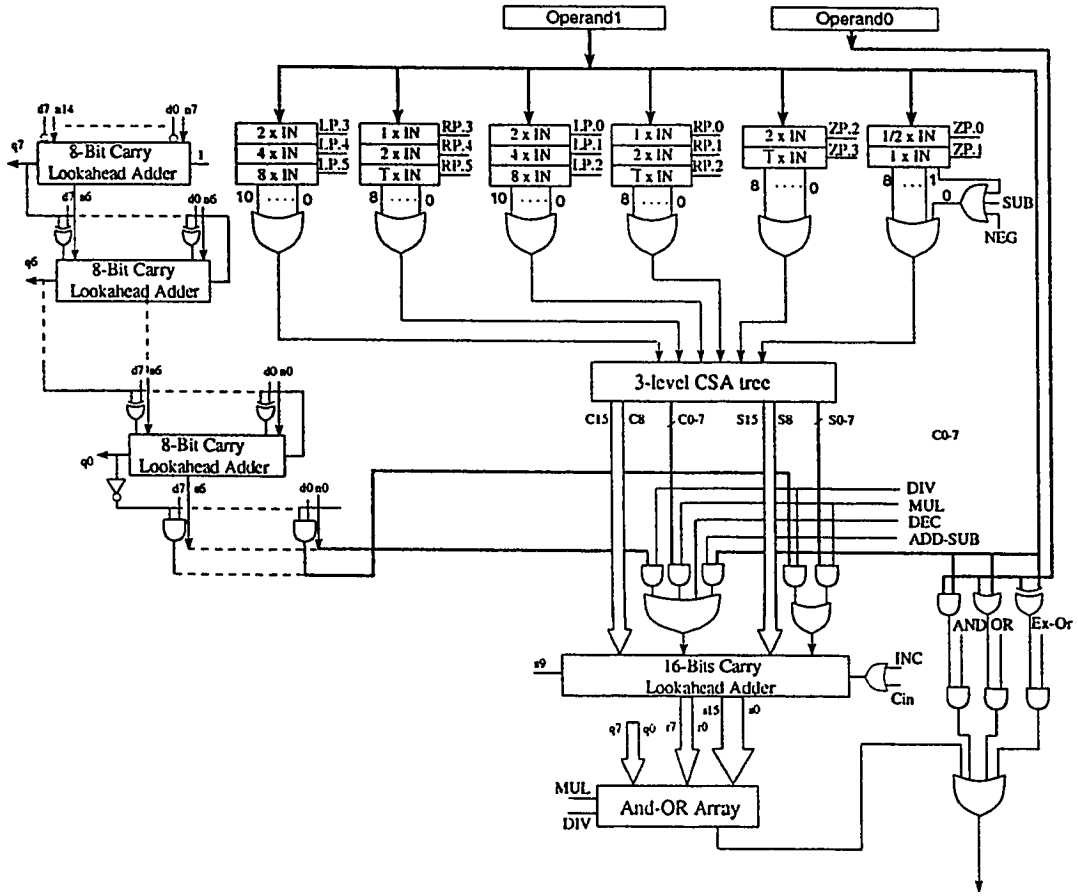

Figure 3.11: Logic diagram of the ALU.

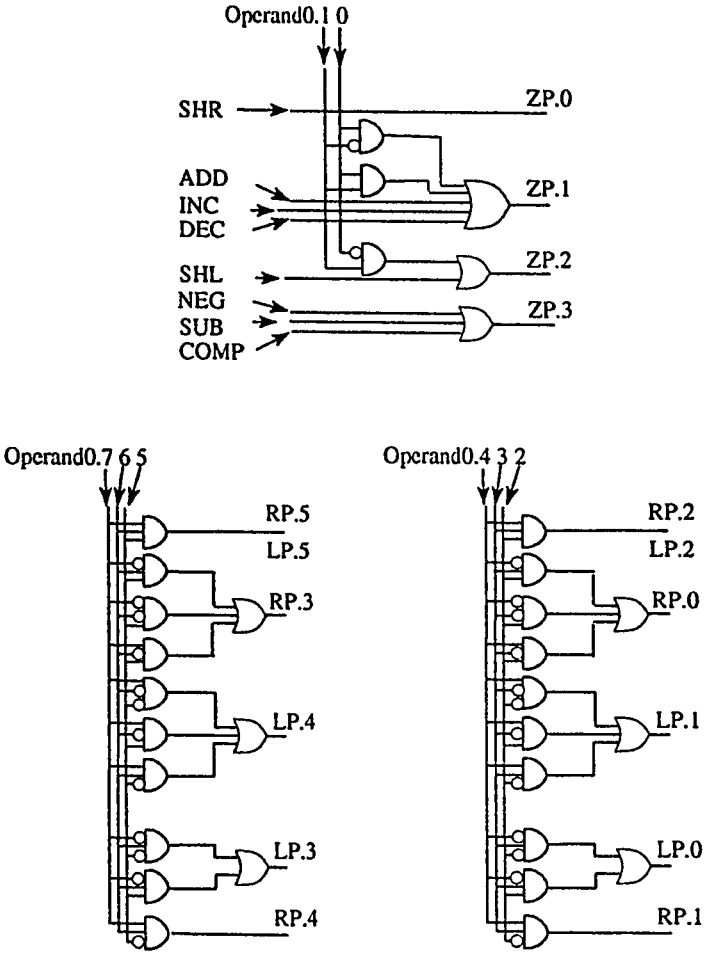

Figure 3.12: ALU encoder.

through out the program execution.

As shown in the Figure 3.13, a 16-bit bus from the host is connected to each word of instruction memory. This bus is used to load the instruction memory with the instructions, during the initialization phase. The instruction address is selected by a 3-bit address $Add - 0$. The host-computer supplies this address to a 3×8 decoder. The output of this decoder is controlled by an AND gate, which activates the instruction write decoder only when input *Initialize* is 1 and input *Data/Address* is 0. When both these conditions are true during initialization phase, then the data is written into the instruction memory (selected by the decoder) during the first phase of the machine cycle (each machine cycle is made up of four clock cycles and divided into four non-overlapping phases).

The outputs of all the instructions are multiplexed using AND/OR logic as shown in Figure 3.13(b), although *Tristate* buffers can be used too; but the tristate buffers present in the library of OASIS require a large amount of chip area. The 16-bit output of the instruction memory is applied at the input to the *Controller* unit (discussed in Section 3.9).

For instruction selection/execution initially a 3-bit program counter was used which sequentially pointed to each instruction. The output of this counter was *ANDed* with the dataflow control unit output. If all the operands for an instruction were available and the counter also pointed to the same instruction then the instruction was executed. Otherwise the counter jumped to another location without doing anything in that cycle. This design degrades the performance of the processor very

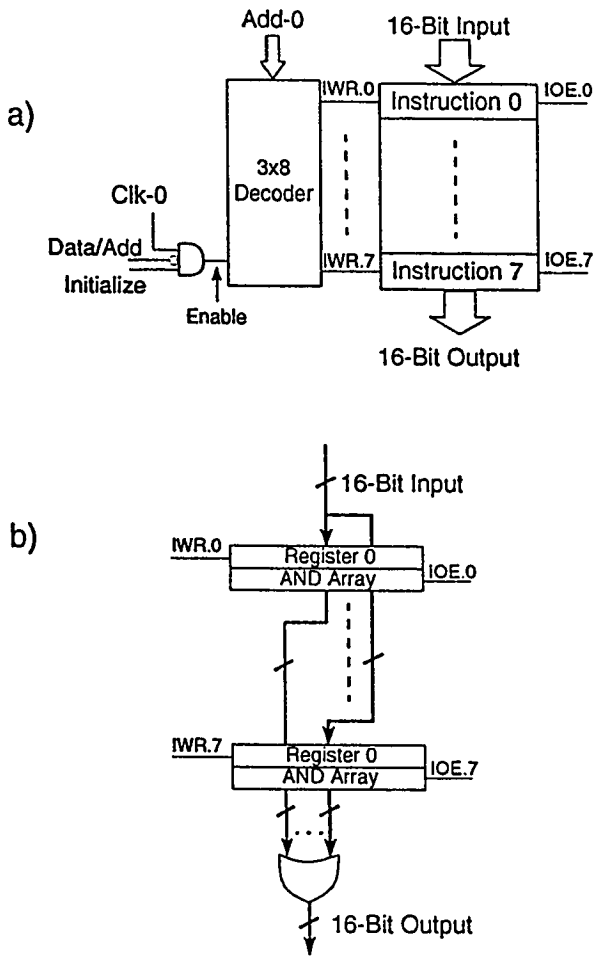

Figure 3.13: Block diagram of instruction memory.

much. Consider, for example, a processing element having only one executable instruction. This counter will cause a delay of seven cycles before the same instruction is re-executed on a new set of operands.

To solve this problem and enhance the performance of the processor array the program counter has been replaced by an *Execution Control* unit based on a priority encoder. This design gives a true dataflow realization to this processor. It allows to skip those instructions which are not ready for execution or do not contain executable instructions. It enables the immediate execution of the same instruction if new operands for that instruction are available. The complete design of this execution control unit is discussed in Section 3.6.

Each instruction has a flag attached to it, which shows the status of the instruction. The flag is set when an instruction needs to be executed and is reset after the execution of the instruction. These flags together with the priority encoder help to execute an instruction. If the flag is reset than the priority encoder puts that instruction at the the lowest priority, and considers only those instructions for execution whose *Execution* flags are set.

3.5.2 Data Memory

Each processing element has eight 8-bit registers for data manipulation. These registers hold the operands needed in the execution of the instruction and also store the results obtained after execution of the instruction. They also help in the communication with its neighboring PEs as shown in Figure 3.14. Each PE is

connected to its eight immediate neighbors using these registers. The data transfer between processing elements via these registers takes place in a single clock cycle.

As shown in Figure 3.16 the data registers are connected to three input buses and three output buses of 8-bits each. Two of the three input buses are connected to the 16-bit *Host* input bus and to the ALU output through a multiplexer. A double input bus is used:

- to reduce the load on the external host data bus, and
- to transfer 16-bit ALU data to two registers in a single clock cycle.

By using a double bus it is possible to transfer data to two registers in a single clock cycle. This helps in transfer of data from host to PE (during initialization and Load instruction) and in single cycle execution of Multiply, Divide, Exchange, Add, and Subtract instructions.

During the initialization of processing element the register address is supplied by the host through two 3-bit address fields *Add_0* and *Add_1*. These addresses are applied at the input of two 3×8 decoders (one is used for the data of *BUS_1* and the other one is used for *BUS_2* data). The output of these decoders is controlled by an AND gate. This AND gate is used to activate the decoders when input *Initialize* is 0 and input *Data/Address* is 1. The host data is written on the registers (selected by the decoders) during the first clock cycle.

During normal program execution, a register is selected with the help of a destination address embedded in the instruction. The data is written into the registers

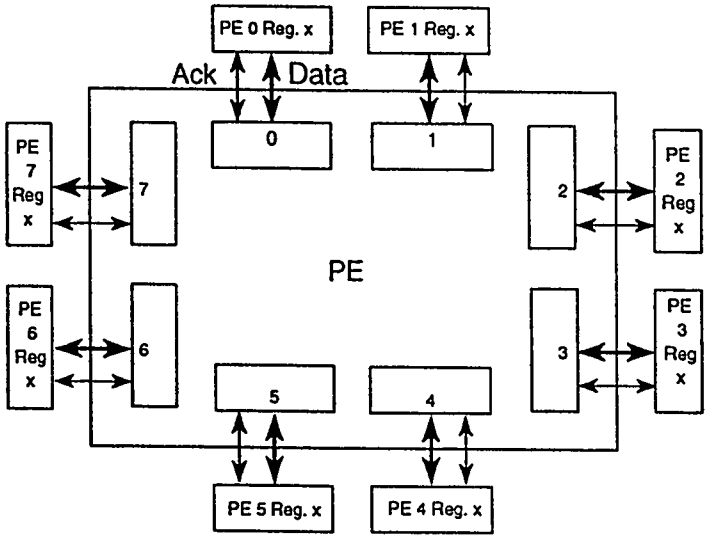

Figure 3.14: Processing element data registers inter-connections with the adjacent PEs corresponding registers.

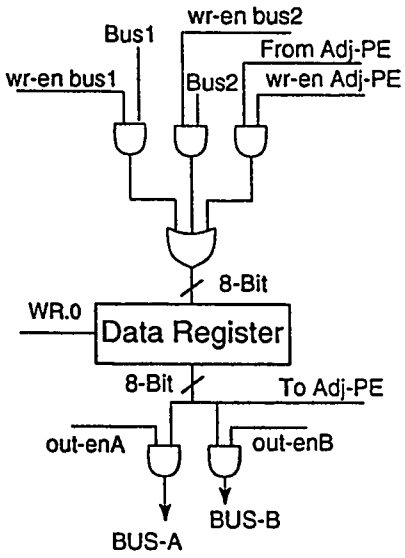

Figure 3.15: Schematic of a 3-input 3-output data register.

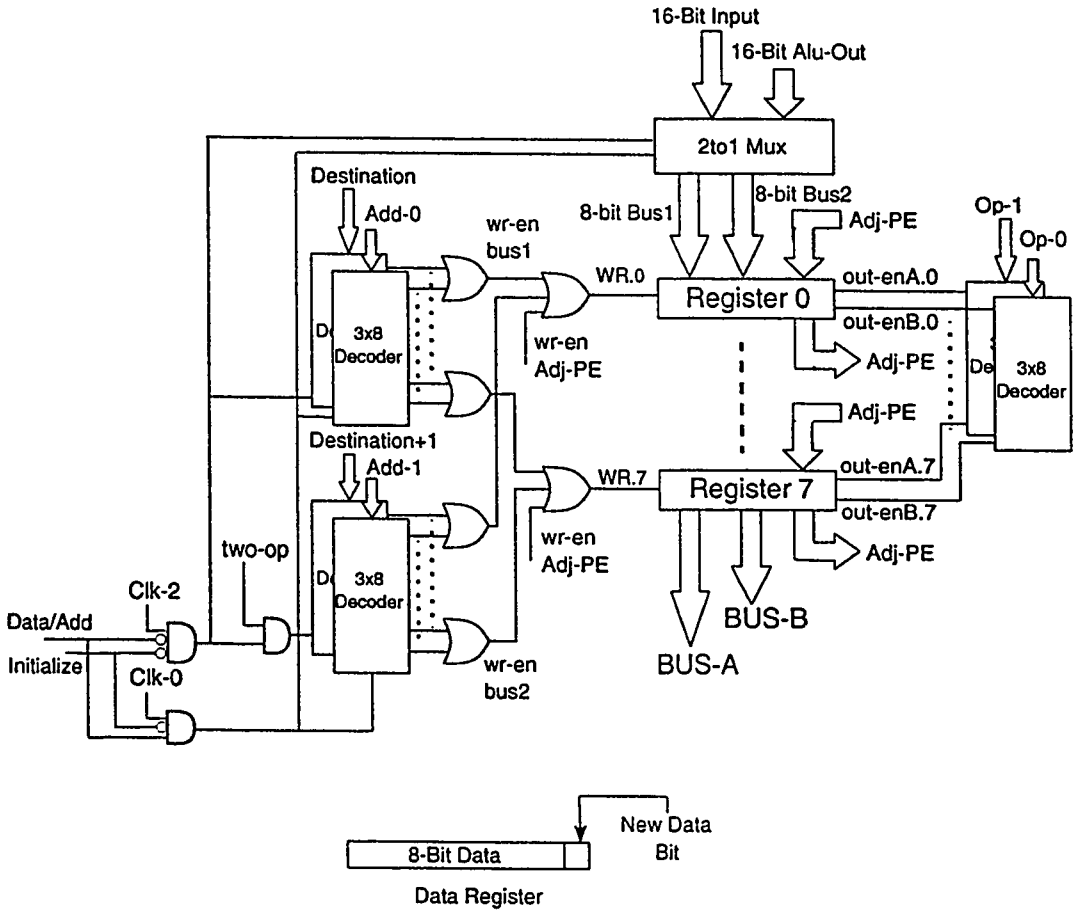

Figure 3.16: Block diagram of the data registers and decoding circuitry.

Each data register has attached to it a *New_data* bit which indicates the arrival of new operands, and the register's willingness to receive new operands from the neighboring PE register. The *New_data* bit is set when a new data is written into the register, and is reset after the consumption of data operands. The *New_data* bits are applied at the input of the *parallel dataflow control unit* (PDCU), which controls the program execution accordingly.

The third input bus connects each register directly to the output of the corresponding register of the adjacent processing element. Similarly, the output of each register is connected to the input the corresponding register of the adjacent processing element, just like a ring as shown in Figure 3.17. This results in tighter coupling and faster communication among processing elements. The data transfer among adjacent processing elements takes place in only one clock cycle and is controlled by a *data transfer control register* (DTCR).

To ensure correct sequencing and data transfer between adjacent PEs, handshaking protocols must be adopted to synchronize the operations. There are two types of asynchronous schemes [2]: the one-way control scheme and the two-way control scheme. In the one-way control scheme, the sender sends data without waiting for the acknowledgment signal of the receiver. This method is suitable for an array processor only when large buffers are provided at the receiver. The two-way control scheme, usually known as handshaking, is preferable for most of the array processors. A proposed handshaking circuit is shown in Figure 3.18. The design uses only one flip-flop and four gates. Although the system clock triggers the flip-flop, the circuit

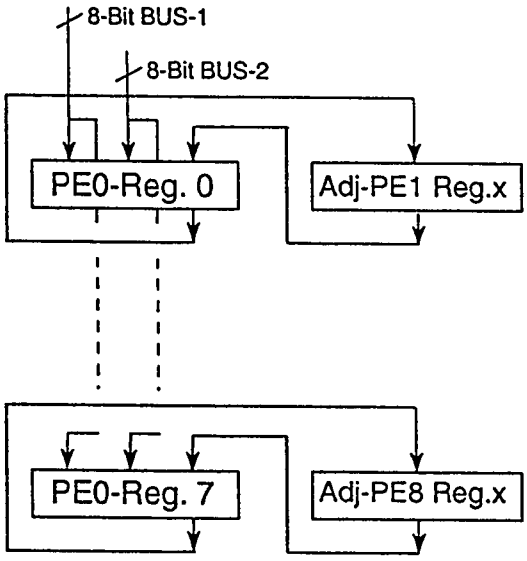

Figure 3.17: Third input of the data registers.

works asynchronously with the clock of the adjacent PE. The circuit performs as follows:

1. Initially all ACK signals are '0'. When data is written into a register ($WR = 1$) then DS is set on the second phase of the system clock.

2. If a *Data_transfer_control* bit (in the DTCR) corresponding to the same register is set, the a *Data_Ready* (DR) signal is produced for the adjacent processing element.

3. In the adjacent PE DR signal is $ANDed$ with the *New_Data* bit of the corresponding register of adjacent PE. A new data bit is set (if the register contains a valid data) then a high acknowledge signal is transmitted to the data transmitting PE indicating that the data is not accepted. The acknowledge signal is also input to the 'parallel dataflow and instruction execution control unit'. This unit does not execute those instructions which write on the data transmitting register (static dataflow execution).

4. As soon as the *New_data* bit of the *data receiving register* is reset, data is written on that register, and the acknowledge signal goes low, indicating data is transmitted properly.

Although the connections between adjacent processing element registers exist all the time, the actual data transfer between them takes place only when a particular bit in the DTCR is set. This is under the control of the programmer. With the

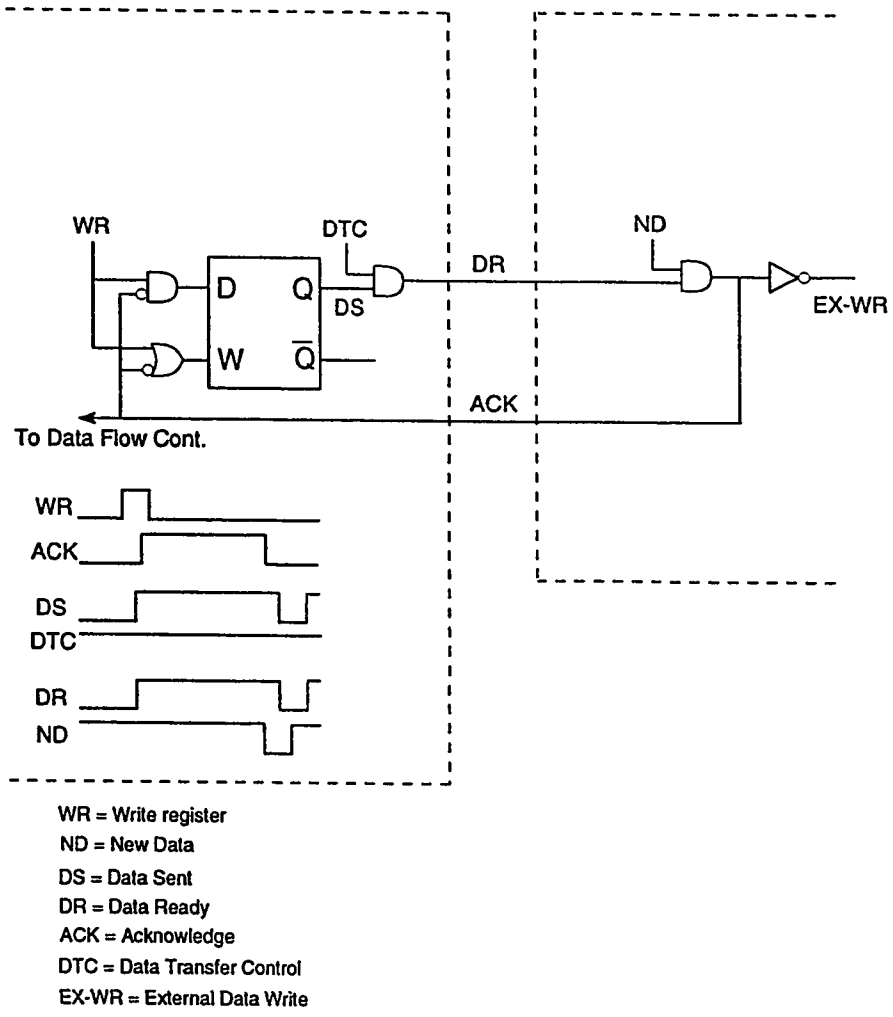

Figure 3.18: The proposed handshaking circuit, and timing diagram.

bit in the DTCR is set. This is under the control of the programmer. With the help of this reconfiguration it is possible to implement any dataflow graph on this processor array. The topology can be reconfigured to star, pyramid, or binary tree, depending on the structure of the algorithm.

3.5.3 FIFO RAM

In order to transfer data to more than one PE a '*Split*' instruction is used (for dynamic data flow execution). This instruction broadcasts the data on the $Host/PE$ data bus with a tag attached. The tag is formed by attaching to the *SourceRegister* number with the *PE identification number*. The processing elements which need this data, accept the data and then process accordingly.

Since the traffic on the bus is higher than anywhere else in the processor array, FIFO RAM (see Figure 3.19) is used to hold the data before they are being transmitted on the bus. The Split instruction transfers the data to the FIFO RAM in a single cycle, and then in subsequent cycles the data is data transferred to the bus, in parallel with the normal program execution. As soon as the FIFO RAM receives new data it sends a $Bus - Request$ to the $Global - Network\ Controller$, which transfers the bus control to the processing element accorded the highest priority. After getting control of the bus, this processing element transfers its data to the bus in a single clock cycle, and updates the FIFO RAM in the next cycle. Each processing element has a FIFO RAM of 8×11 bits. This can be used for other purposes depending on the program.

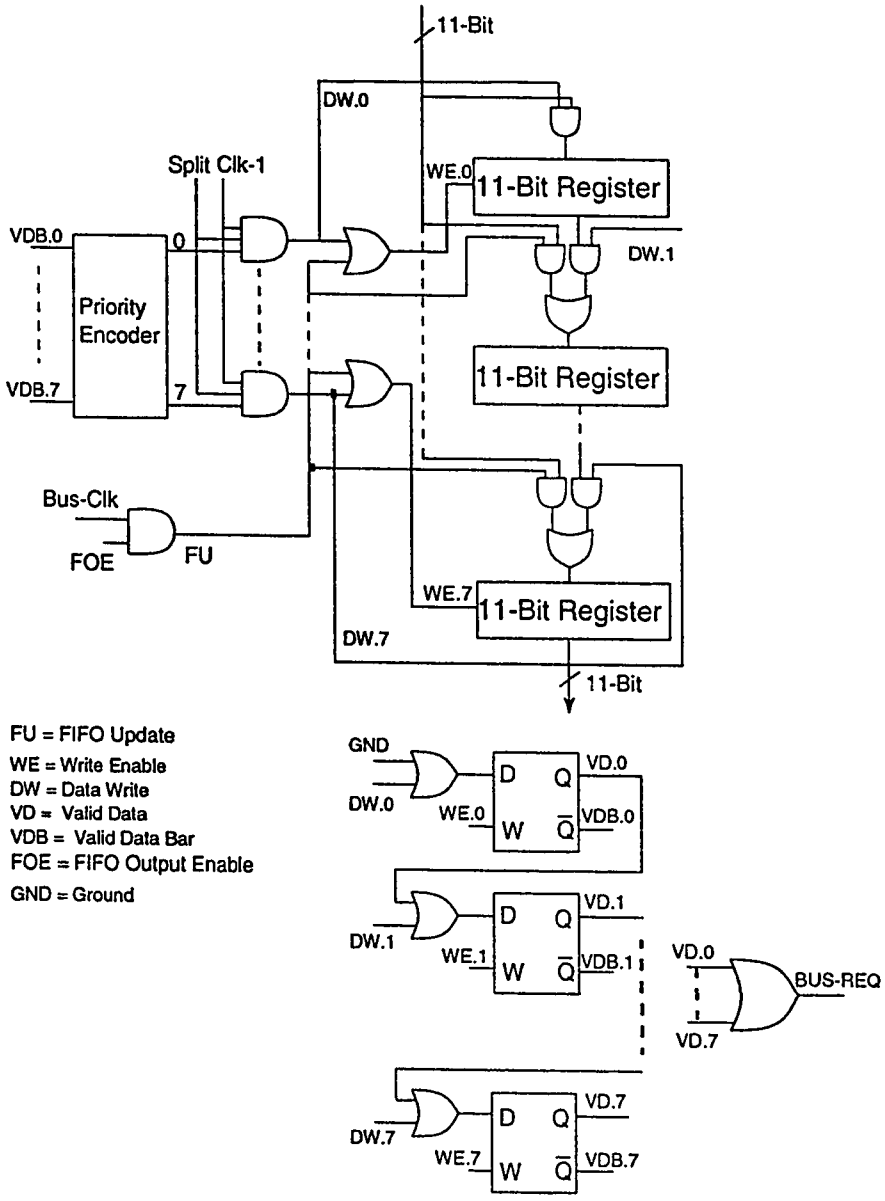

Figure 3.19: FIFO RAM and its control.

3.6 Parallel Dataflow and Instruction Execution Control Unit (PICU)

The PICU is a uniquely designed block that controls the dataflow operation of the processing element. The instructions in the processing element are not executed in any predetermined order. Instead, the arrival of all the operands for a certain instruction enables the execution of that instruction. There are two types of flags which monitor the data movement (among registers) and instruction execution.

The flags that monitor the data movement in the data registers are named *New_Data* flags, they are made up of D-flip-flops. Each data register has a *New_Data* flag attached to it. For each data register, a flag indicates whether the register has a valid operand or can receive a new operand. If a flag corresponding to a register is set, it means that the data in that particular register is valid and can be used for instruction execution. The reverse is true when the flag is reset. This unit continuously and concurrently monitors the *New_data* flags and looks for instructions that need operands for their execution.

The flags that monitor the instruction execution are named *Execute* flags. Each instruction has an execution flag (see Figure 3.22). which resets after the execution of the instruction. These flags are used by the instruction sequencing unit for the execution of one instruction at a time.

By the help of *New_Data* and *Execute* flags, acknowledge signals from adjacent processing elements and data transfer control configuration register, PICU controls

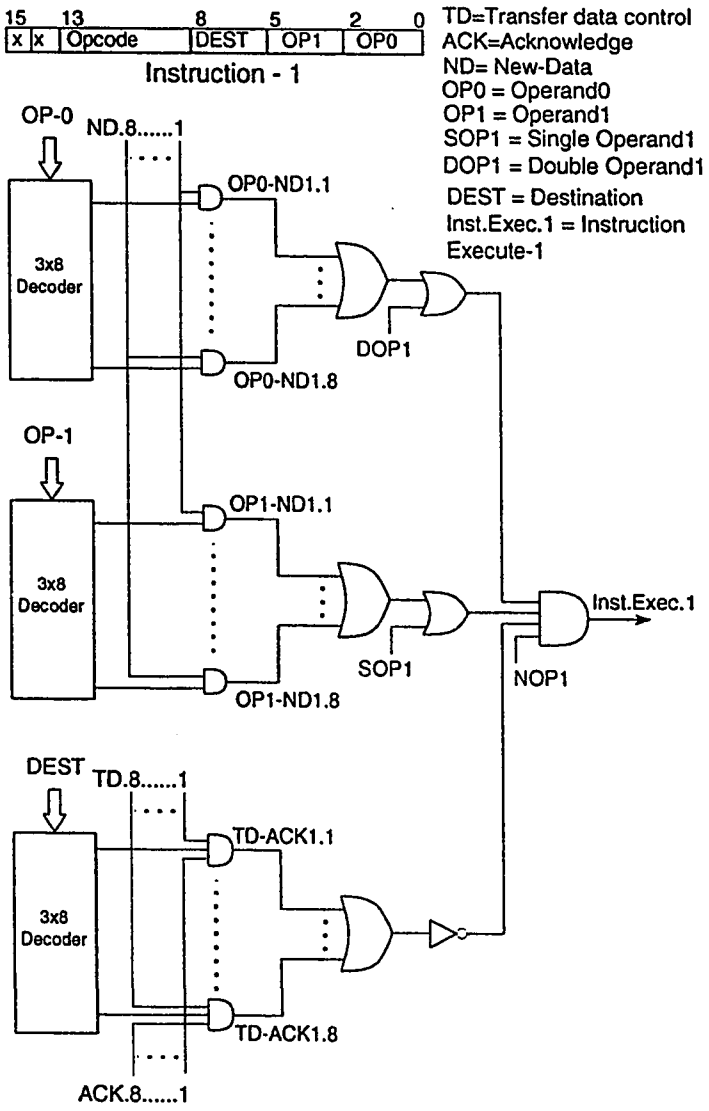

Figure 3.20: Dataflow Control unit for a single instruction.

the instruction execution. The parallel dataflow control unit (PDCU) works as follows (see Figure 3.20).

All instruction outputs are applied to this unit. The three bit fields of *Operand*0, *Operand*1, and *Destination* are applied separately to three (3 × 8) decoders. The outputs of *Operand*0 and *Operand*1 decoders are ANDed with the *New_data* flags. To detect the arrival of operands for instruction execution, the AND gate outputs are ORed to implement the following equation:

$$O = \left[\begin{array}{ccc} Operand.7 & \ldots & Operand.0 \end{array} \right] \left[\begin{array}{c} New-data.7 \\ \vdots \\ New-data.0 \end{array} \right]$$

The five bit opcode field is applied to a simple decoder which detects the valid opcode, single operand and double operand instructions, and generate NOP, SOP, and DOP signals respectively.

The output of destination decoder is ANDed with the data transfer control register output and adjacent PE acknowledge bits. The outputs of these AND gates are NORed to implement the following equation for *Static dataflow* execution.

$$\overline{D} = \left[\begin{array}{ccc} Destination.7 & \ldots & Destination.0 \end{array} \right] \left[\begin{array}{c} Transfer-data.7 \\ \vdots \\ Transfer-data.0 \end{array} \right] \left[\begin{array}{c} Acknowledge.7 \\ \vdots \\ Acknowledge.0 \end{array} \right]$$

When the NOR gate output is zero the instructions which writes on the data transmitting register is disabled. This happens only when the acknowledge data bit corresponding to the set transfer data bit is one, indicating that the data has not been consumed by the adjacent PE receiving register

The PDCU performs the same operations on all the instructions concurrently, as shown in Figure 3.21 and generates eight output signals named *Execute_ Instruction*.1...8. A high *Execute _ Instruction* signal is generated when all the following conditions are true.

1. When *Operand*0 of an instruction has arrived (in case of a 1-operand instruction).
2. When both operands have arrived (in case of 2-operand instruction).
3. The acknowledge signal corresponding to the set bit of the DTCR is zero.
4. The operand code represents a valid instruction.

By using PDCU it is possible that many instructions get ready for execution at the same time, depending on the arrival of the operand. Parallel execution of all the instructions is not possible due to resource limitations. Therefore the output of the PDCU is applied to an execution circuit containing a *priority encoder* (see Figure 3.22). This circuit enables one instruction at a time. The *Execute_Instruction* signals of the dataflow control unit are ANDed with the instruction *Execute* flags and applied to the input of the priority encoder, which selects the highest priority instruction for execution. The instruction *Execute* flag resets after instruction execution and is set again after all the instruction *Execute* flags are reset (i.e., if all the instructions are executed) or the priority encoder output is zero (i.e., if no instruction is ready for execution). By doing so, an instruction is not executed again (even if it has received all its operands) unless all the instructions get executed or no other instruction is ready for execution.

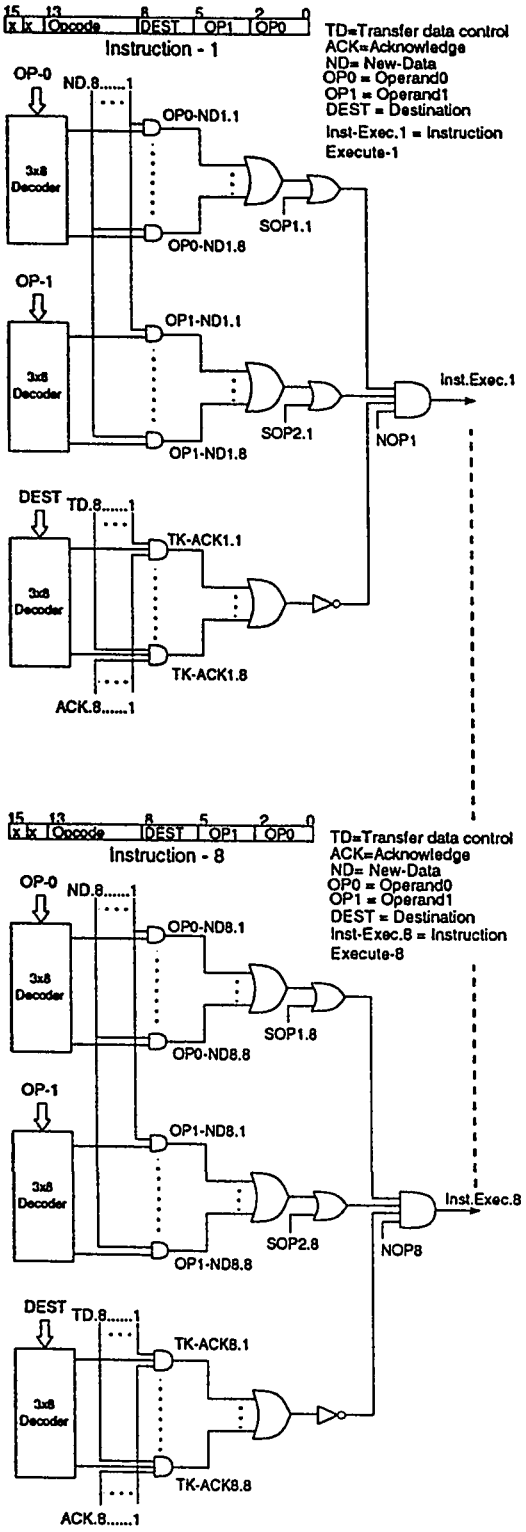

Figure 3.21: Parallel dataflow control unit for eight instructions.

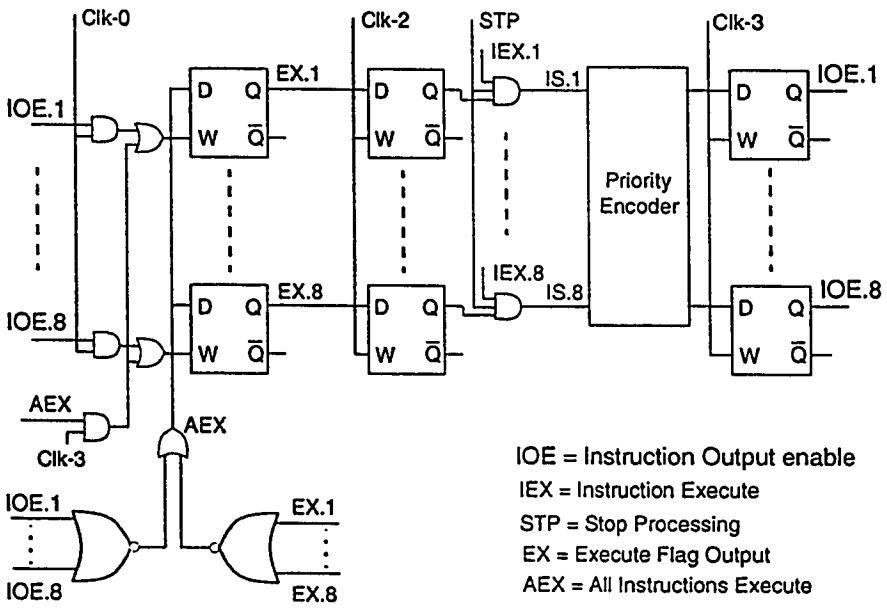

Figure 3.22: Execution unit logic diagram.

3.7 Configuration Registers

The 'Configuration Registers' give the reconfigurative characteristic to this processor array. There are three configuration registers (see Figure 3.23). The *Data transfer control register* (DTCR) is used to control the transfer of data between the registers of adjacent processing elements. If bit i of this register is set, data is transferred between PE register i and its neighbor PE register (to which PE register i is connected). The transfer takes place solely by a simple handshaking protocol as discussed in Section 3.5.2, without any software control.

The *Host data request register* (HDRR) helps to load data from the memory of the Global Network Controller (GNC) into the required register. Once the data of the register is consumed (indicated by the *New_Data* bit attached to each register), a request for new data is sent to the GNC, if a bit corresponding to that register is set in the host data request register. This request is sent, during the same machine cycle data is consumed. In the subsequent cycles the data is loaded into the data register, in parallel with the normal program execution. This technique is used to avoid the lengthy delays associated with load instructions.

With the help of the *PE Identification number configuration register* (PINCR), a processing element can be assigned with any identification number. This identification number is used as a tag in dynamic data flow execution, and in communication with the host. This feature makes the design very attractive for VLSI implementation because all the PE's are exactly identical; therefore only one mask can produce any numbers of PEs. This configuration register makes fault tolerance very easy. Upon determination of the PE failure, a spare PE can be loaded with failed PE identification number, without the

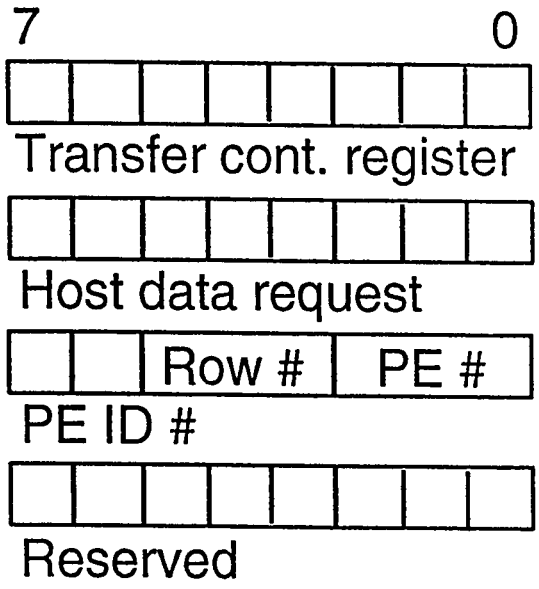

Figure 3.23: PE configuration registers.

need of remapping the program to the new set of PE's excluding the failed PEs as done by [27].

As one can see from Figure 3.23 there is also a fourth register which is still unused, this register can be used for future expansion. One of the possible use of this register is holding constants [17]. If bit i of this register is set, data in register i should be treated as a constant and its value should not be changed during program execution.

These registers are configured by the host computer during initialization phase and remain unchanged throughout the program execution. As shown in the Figure 3.24, the input of each configuration register is connected to the host through a 16-bit bus. This bus is used to load these registers with data during the initialization phase. The register address is selected by a 3-bit address *Add_0* supplied by the host computer. This address is applied at the input of a 3×8 decoder, which also controls the writing of data on the *matching unit registers* discussed in Section 3.25. An AND gate is used to activate the write decoder only when *Initialize* input is 1 and *Data/Address* input is 1. When this condition is true *i.e.*, during initialization phase, then the data is written on the configuration registers (selected by the decoder) during the first clock cycle.

3.8 Direct Matching Unit

In this processor a novel and simplified process of matching of tags has been used. The direct matching technique used in this processor is slightly different from the one reported in the literature [7, 28, 29]. But it avoids the expensive and complex process of associative search used in previous dynamic dataflow architectures. The 'Direct Token Matching' unit is used for dynamic dataflow execution of a program. It contains four registers of 11 bits

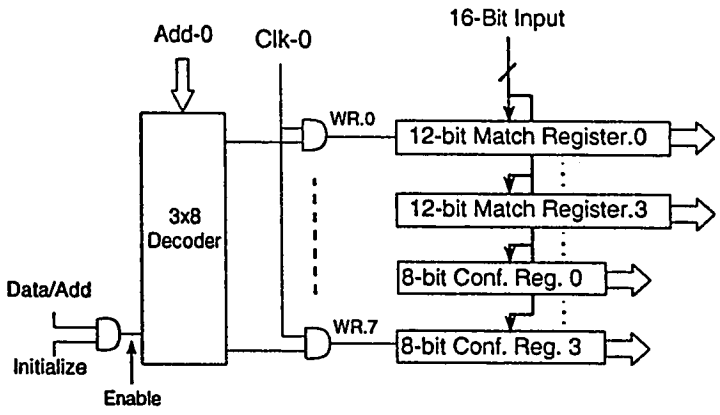

Figure 3.24: PE configuration registers and matching registers input.

each. In these registers the matching tags and destination register addresses are stored during the initialization phase. A bit attached to these registers indicates that the contents of the register are valid. A tag is the catenation of a PE identity number, and a source register number. During runtime, whenever a processing element transmits any data on the Bus (using the 'Split' instructionas discussed in Section 3.5.2), all the processing elements on that bus compare the tag of that data with the prestored tag (see figure 3.25). If a match is found, the data is stored in a temporary memory or transferred directly to data registers, otherwise data is discarded. By using a direct matching technique, matching of tags is performed in only one machine cycle. Due to the limited bandwidth of the common BUS, a FIFO RAM is used to hold data for some time.

As shown in the detailed diagram of the matching unit (Figure 3.26), the data along with the tag is written in a 19-bit wide latch, controlled by the GNC. The tag in the latch is matched with the contents of prestored tags in the matching registers. If a match is found then data is transferred to a temporary storage, otherwise the data in the latch is overwritten by the GNC with new data.The data transfer to the data registers is dependent on the *New_Data* bit if the bit is zero (indicating that the register is empty), the data from the storage is transferred to the data registers where it is used by the instructions for program execution.

3.9 Hardwired Control and Instruction Decoder

Three design approaches [22] are generally used for designing hardwired control:

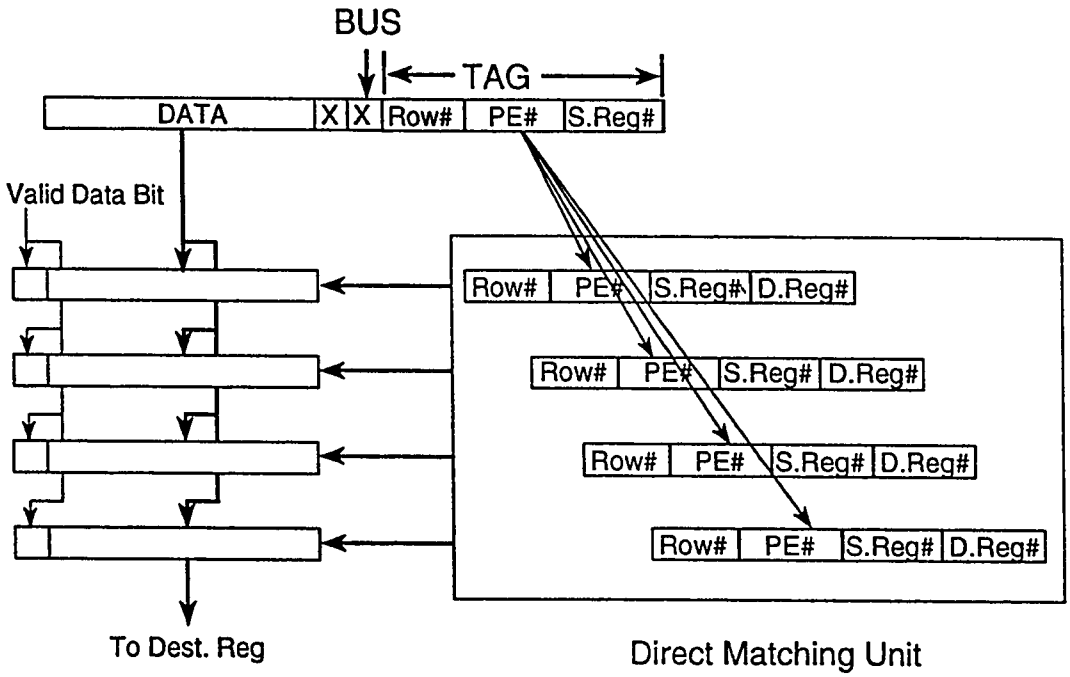

Figure 3.25: Block diagram of direct matching unit of the DF-RISC-A.

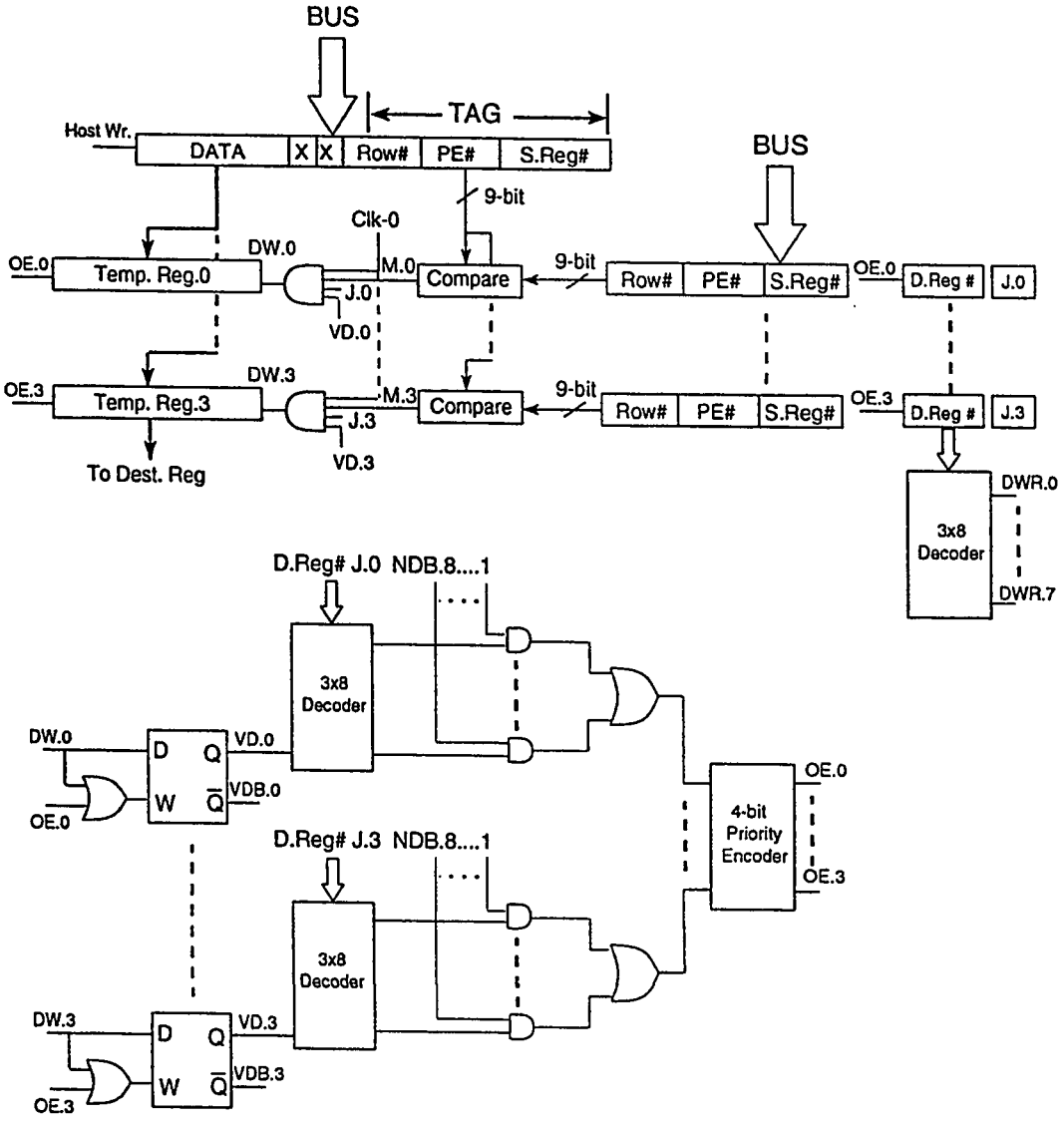

Figure 3.26: Direct matching unit of the DF-RISC-A.

1. The standard approach to sequential circuit design of switching theory, which is called the state-table method, since it begins with the construction of a state table for the control unit.

2. A method based on the use of delay elements for control-signal timing.

3. A related method that uses counters, which we call sequence counters, for timing purposes.

Method 1, the most formal of these design approaches, incorporates systematic techniques for minimizing the number of gates and flip-flops. Methods 2 and 3, which are less formal, attempt to derive a logic circuit directly from the original (flowchart) description of the control-unit behavior. The resulting design may not contain minimum number of gates and flip-flops, but they are often obtained with much less effort. furthermore, these designs are usually easier to comprehend and therefore more likely to be free of error and easier to maintain.

In this processor design, third approach is used to design a simple, efficient, and reliable control unit. The heart of this control unit is a 2-bit counter as shown in Figure 3.27. The input of this counter is a two-phase system clock (counter increments in the second phase of the system clock). The output of this counter is connected to a 2×4 decoder. The counter reset is connected to the resets of all the flip-flops of the processor. The counter cycles continuously through its 4-states (00, 01, 10, 11) decoder generates four non overlapping phases, corresponding to counter outputs. The width of each phase is equal to one clock period. All operation of the processor are then controlled by these four non-overlapping phases.

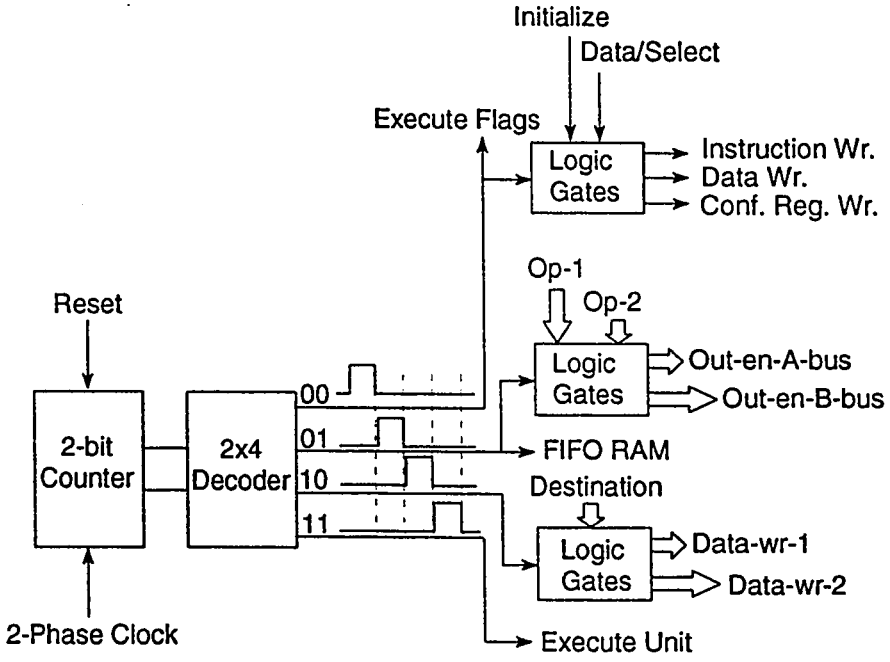

Figure 3.27: Hardwired control unit.

- During the first phase of the machine cycle:

    1. Depending on *Initialize & Data/Address* select bits (Table 3.6) selected instruction is decoded or instruction, data, or configuration & matching register write operations are performed.
    2. Instruction execution flags are updated.
    3. In the direct matching unit if a match is found then the data of the bus (already stored in a latch) is stored on temporary registers.

- During the second phase of the machine cycle, Operands are selected and applied at the input of the ALU, or written on the FIFO RAM.

- During the third phase of the machine cycle:

    1. ALU output and Adjacent PE register data (if available) is written on the data registers.
    2. *ALU flags* are updated.
    3. *Instruction execute* flip-flops are updated according to the execute flags.

- During the fourth phase of the machine cycle, execute flags are updated and instruction is selected for execution.

  As shown in Figure 3.28 a simple instruction decoder [30] has been designed.The Instruction-opcode is applied to the opcode decoder, the output of this decoder is used for instruction execution. The operand and destination fields are applied to separate decoders, which generate the addresses of the data registers.

Table 3.6: Combinations of *Initialize* and *Data/Address select* bits.

Initialize	*Data/address*	*Operation*
0	0	Normal program execution
1	0	Instruction write
0	1	Data write
1	1	Configuration register write

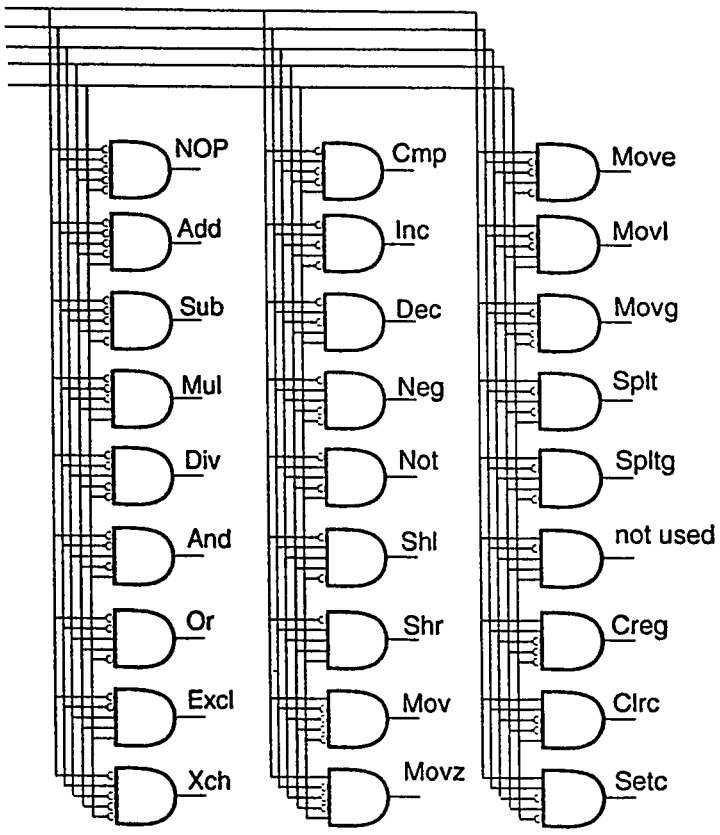

Figure 3.28: Instruction decoder.

Chapter 4

Array topology and Design environment

4.1 Array Topology

When designing of a large multiprocessor computing system one of the primary design decisions involves the topology of the communication structure amongst the processors. In the past the designers have restricted their attention to simple inter-connection topologies such as linear arrays, two-dimensional arrays and circular rings [31]. More recently, with the improvement of VLSI technologies, a number of projects have been initiated with the aim to design large multiprocessor systems with more sophisticated inter-connection topologies.

As this processor has been designed for arbitrary algorithms, special attention has been given to array topology design. Based on VHDL simulations a topology for processing

element interconnection has been designed. Figure 4.1 shows the floor-plan of the chip.

In the current prototype, four PEs are arranged as a square array, with four PEs per row (this can be extended to 16 PEs per row). Each PE has a 6-bit unique address, assigned by the programmer. In order to facilitate maximum communication between PEs, each PE can communicate with its 8 immediate neighbors through the boundary registers/ports, while it can communicate with the other PEs and the host using the *Global network controller*(to be discussed in Section 4.2) and the host bus which runs between two alternate rows of PEs.

Although the connections between adjacent PE registers exist all the time, the actual data transfer between them is controlled by a single bit in the '*Data Transfer*' configuration register, which is under the control of the programmer. There are three possible ways of data transfers between PE registers (see Figure 4.1):

1. Data transfer from PE0 to PE1.

2. Data transfer from PE1 to PE0.

3. No data transfer between PE0 and PE1.

The data transfer between processing elements is controlled by a simple handshaking protocol as discussed in Section 3.5.2.

This topology gives maximum flexibility to the programmer and results in tighter coupling and faster communication among processing elements. This is a fully connected graph, which can be configured to any topology depending on the algorithm. Since the topology can be reconfigurable, it is possible to implement any dataflow graph on this processor array. The topology can be reconfigured to star, pyramid, or binary tree, depending

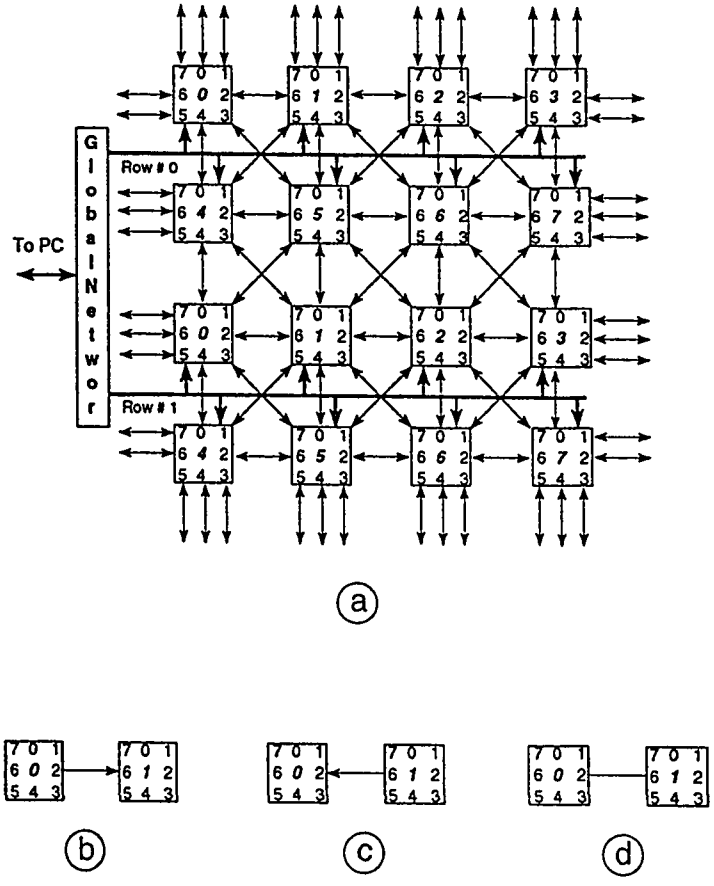

Figure 4.1: Floor-plan of the DF-RISC-A.

on the structure of the algorithm.

4.2 Global Network Controller

The 'Global Network Controller' (GNC) takes care of the communication between PEs and host. It generates control signals for data transfer between the host and PEs. It receives 16 bits of input data from the host and processed data from PEs. This network controller is used to interface the processor array with the parallel port of a Personal Computer. Additional PEs can easily be added due to modular nature of the communication network, and facilitates the future design of a massively parallel computer.

As shown in Figure 4.2, the GNC contains a bus resolver and a memory management unit, a 64k-byte memory, and a handshake unit. This unit has not been included in the VLSI chip of the array processor due to the following reasons:

- It requires a large amount of memory for data storage, which can be easily included in the system using off-the-shelf memory chips, instead of using the real estate of the processor chip.
- An external communication controller makes it possible to combine several processor chips, for designing a massively parallel computers.
- An external communication controller helps to probe the communication between the processing elements and network controller for debugging purposes.
- The host communication unit can be modified according to different computer systems of different vendors (at present it is designed for a 486-personal computer parallel port).

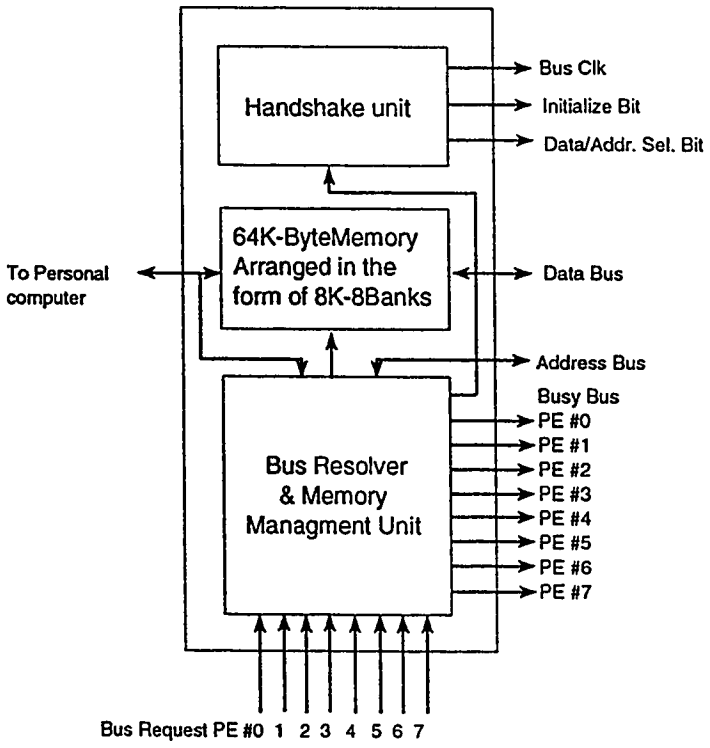

Figure 4.2: Global Network Controller block diagram.

The Global Network Controller works as follows. During the initialization phase it selects a PE for input, according to the priority. Then it generates the *Initialize* and *Data/Address select* signals. Then it sends over the address bus the address for the instruction, data register and matching & configuration registers, and on the 16-bit data bus it sends data.

When a PE requests for a data transfer, by using the *Bus Request* signal to the Global Network Controller, then, depending on the priority of the PE, GNC transfers the control of the bus to the PE, using *Busy Bus* high signal (only one *Busy Bus* signal can be activated at a time) and writes the contents of Bus into the latches of the PEs, using *Busy Bus* and *Bus Clk* signals. The global network controller also writes the same data into the global network memory, which can be accessed by the host computer.

4.3 Regularity, Modularity and Inter connectivity

Some of the desirable features of architecture which facilitate implementation in VLSI include modularity, regularity, and local inter-connectivity. Systolic architectures exhibit these characteristics, as they are highly asynchronous [1] therefore they suffer with problems discussed in Chapter 1. These problems are solved by the dataflow computers. But most of the dataflow computers that are built to date are designed for complex operations [6, 8, 7, 9], therefore it is not fair to compare these processors with this low complexity processor. However the one proposed by [10, 3], can be compared with this processor. The comparison is given in below.

4.3.1 Regularity

In [10, 3] each PE has its own fixed identification number, which makes that PE different from the others.

In this design the PE identification number has been assigned during the reconfiguration/ initialization phase (using the PE ID, configuration register). The PE's are addressed through a single select line, which is controlled by the GNC. This not only makes all the PEs exactly identical, but also reduces the hardware for matching the PE address and highly suitable for fault tolerance.

4.3.2 Modularity

In [10, 3], rows of PEs are added to increase the array size, which eventually results in a bottle neck on the host communication bus.

In this design an array of PEs is controlled by a GNC, as discussed in Section 4.2. This combination of PEs and GNC makes a module, and several such modules can be combined together to make a larger array. All these modules can communicate each other and host through the GNC as shown in Figure 4.3.

4.3.3 Inter connectivity

In [10, 3], each PE is connected to six of its immediate neighbors, while in this design each PE is connected to eight immediate neighbors. This high degree of connectivity of the topology and reconfiguration simplifies the task of mapping of arbitrary graphs on the array.

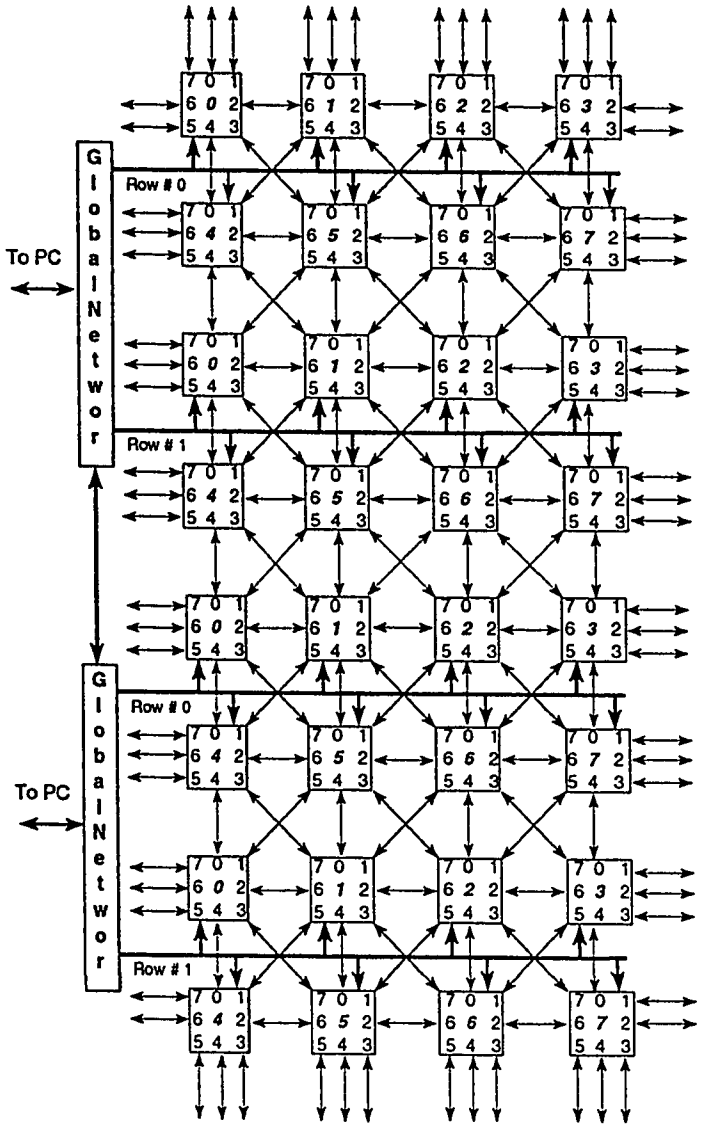

Figure 4.3: Interconnection of two modules.

4.4 Design Environment

A typical process for the design of a digital system is shown in Figure 4.4. Initially, the hardware designer starts with a design idea, then implements that design at the behavioral level. Based on the intended behavior of the system, a procedure is developed for controlling the data between registers and logic units through buses, this is called data path design [32].

Logic design is the next step in the design process, and involves the use of primitive gates and flip-flops for the implementation of data registers, buses, logic units, and their controlling hardware. The result of this design stage is a netlist of gates and flip-flops.

The next stage is the replacement of gates and flip-flops by transistors. This stage also produces a layout for fabrication.

The final step in the design is manufacturing, which generates from the transistor layout several masks for IC fabrication.

4.4.1 VHDL Simulations

Simulators are used to verify each design stage. In addition to a circuit description, a simulator also needs a set of simulation data or stimuli. The simulation program applies this data to the input description at the specified times and generates responses of the circuit. The results of a simulation program may be illustrated by waveforms, timing diagrams, or time-value tabular listings. The designer interprets these results and determines whether to modify a design stage if simulation results are not satisfactory. Simulators can be used at any design stage. At the upper level of the design process, simulation provides information regarding the functionality of the system under design. Simulators normally run

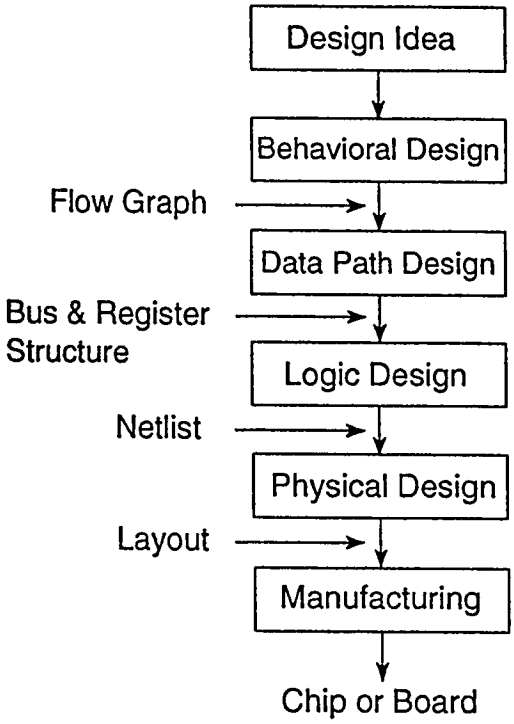

Figure 4.4: Steps involved in a VLSI design process.

very quickly to verify basic functionality. Simulation at a lower level of design process, for example gate level or transistor level, runs much more slowly, but provides more detailed information about the timing and functionality of the circuit. To avoid the high cost of low level simulations runs, simulators should be used to detect design flaws at the early design stages.

This processor was first modeled in VHDL (Very High Speed Integrated Circuit Hardware Description Language) at the behavioral level. One of the many advantages of VHDL is that it supports concurrent execution of processes [32, 33, 34] and statements, which is very helpful in designing a parallel processor.

The model used for VHDL simulations is shown in Figure 4.5. It uses the same instruction format as proposed in [10]. The two register banks (Bank0 and Bank1) are used for holding the operands for the instructions and their results. The registers of Bank0 located on the periphery of the PE, are directly connected to the corresponding registers of the adjacent PE's. The registers of Bank1 are used to temporarily hold the data from host or the results of execution unit, and operands from adjacent PEs, while Bank0 holds the operands required by different instructions. In this model the result of an instruction is transferred to a $Bank0$, before transferring to the $Bank1$. A program counter of three bits is used which executes the instructions in coordination with dataflow control unit as discussed in Section 3.6. To ensure correct sequencing and data transfer between adjacent PEs, one-way synchronization scheme (the sender sends data without waiting for the acknowledgment signal of the receiver) as discussed in Section 3.5.2 was used. This model has been tested for a simple expression:

$$x = (a + b) \times (c + d)$$

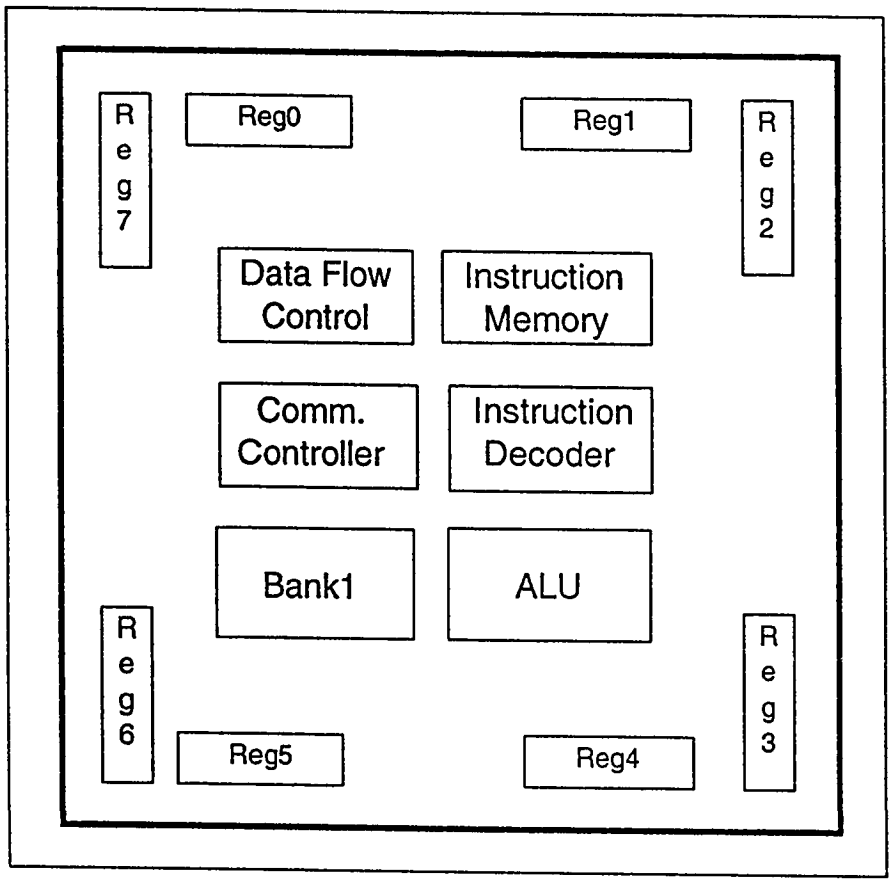

Figure 4.5: Processor model used for VHDL simulations.

The following observations were made after VHDL simulations:

1. The instruction format proposed by Koren is very difficult to implement, moreover, it is difficult to differentiate between first and second operand, and a maximum of two operands can be specified, no specification for the destination operand.

2. The use of program counter degrades the processor performance. Use of this counter results in a seven cycles delay before the re-execution of the same instruction on a new set of operands.

3. The one way synchronization scheme is not useful in dataflow processors. As there is no acknowledge signal for the sender. This scheme results in erroneous response.

4. Without a matching unit it is not easy to extract all the parallelism of an algorithm.

These defects have been removed in the actual implementation of the processor. A high performance processor has been designed as discussed in Chapter 3.

4.4.2 OASIS

The tools that generate layout from VHDL models are not available at the KFUPM. The hardware described in Chapter 3 has been modeled using Logic3, a hardware description language, and simulated to verify functional correctness. The Logic3 description is then input to the OASIS silicon compiler to produce a standard-cell layout in SCMOS technology [23]. The OASIS silicon compiler is used to produce the layout in Magic [35, 36]. To verify the correctness of the layout, and determine the operating speed, the circuit is extracted from the layout and simulated using the switch level simulator *irsim*.

During this stage of design process, modular approach has been used. All the units discussed in Chapter 3 have been designed and simulated separately and combined together to form a processing element. By working at the gate level, it is possible to minimize the chip area.

The layouts generated by MAGIC for different modules are shown in Figures 4.7 to 4.14. The chip area of a complete processing element in 2 μ SCMOS technology is 7512 × 8081 μm^2 and requires 4261 CMOS gates (transistors: $n\text{-}channel$ = 15760 $p\text{-}channel$ = 14873). The specifications of this processor are summarized in Table 4.1. The power consumption as shown in Figure 4.6 has been calculated using CaZM. At 20 Mhz the power consumption of the processor is 810 mW, while at 125 Mhz the power consumption increases to 1.72 W.

Table 4.1: DF-RISC-A processor specifications.

Instructions	25
Instruction length	16
Instruction opcode	5
Data Registers	8x8 bits
Program Memory	8x16 bits
Configuration Register	4
PE Host Comm. Data Bus	16 bits
Transistors	n-channel=15760 p-channel=14873
Gates	4261
PE size	7512 × 8081 μm^2
PE per Row	4
PE per Chip	16
External Data memory	8k bytes per PE
Technology	2.0 μm n-well CMOS (MOSIS).
Machine cycle	4-clock cycles
Clock Frequency	20-Mhz
Power consumption	810 mW

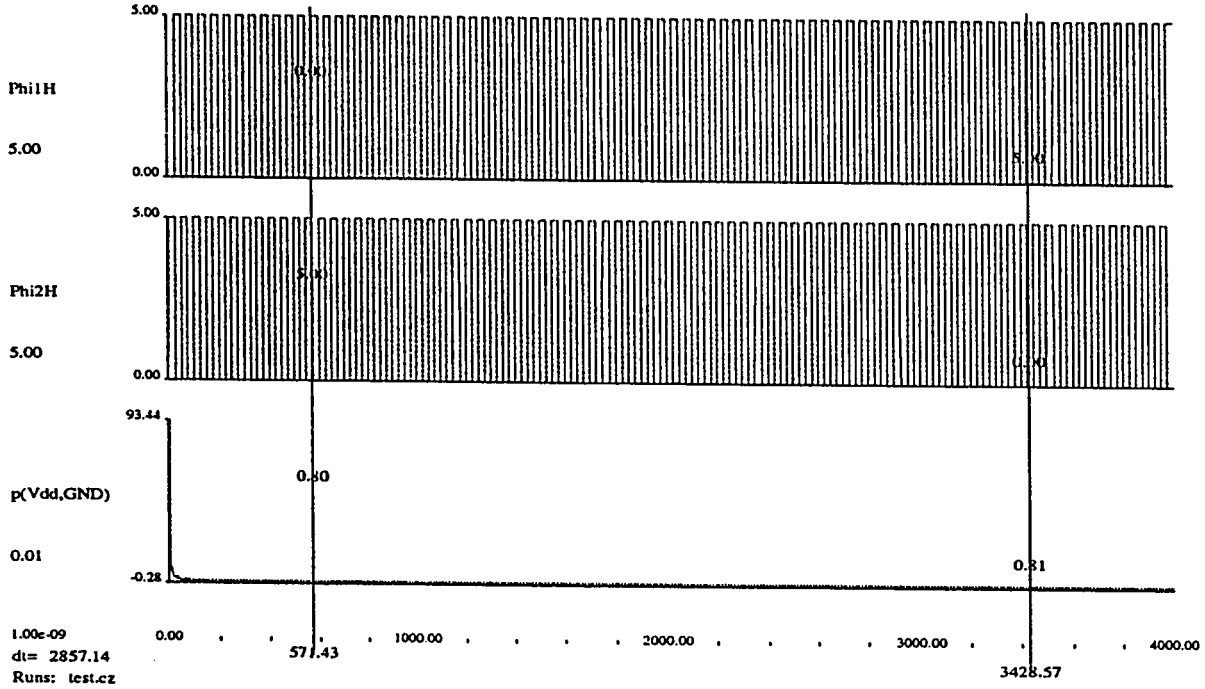

Figure 4.6: Simulations for the calculation of the power consumption of the processor.

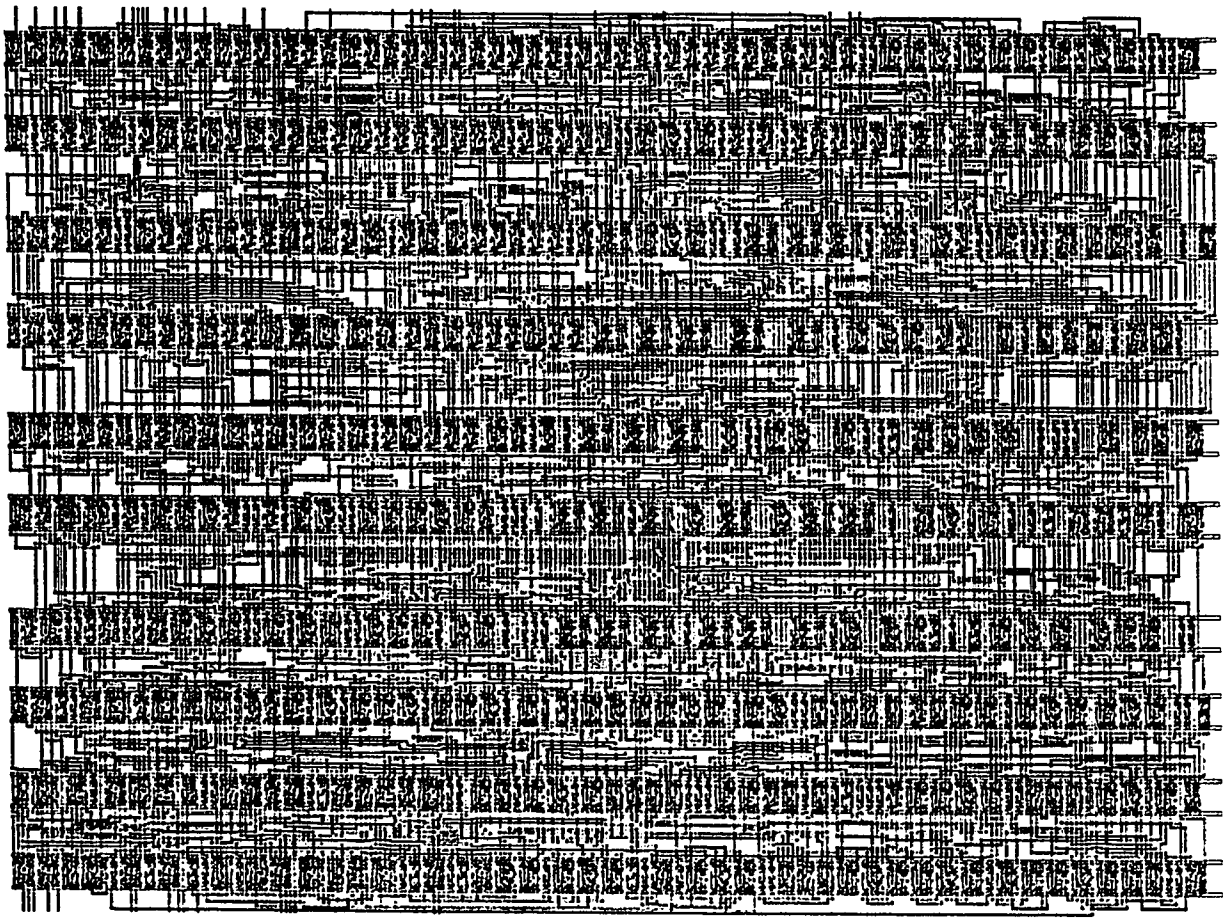

Figure 4.7: ALU layout produced by MAGIC, area = $1467 \times 1984\ \mu m^2$.

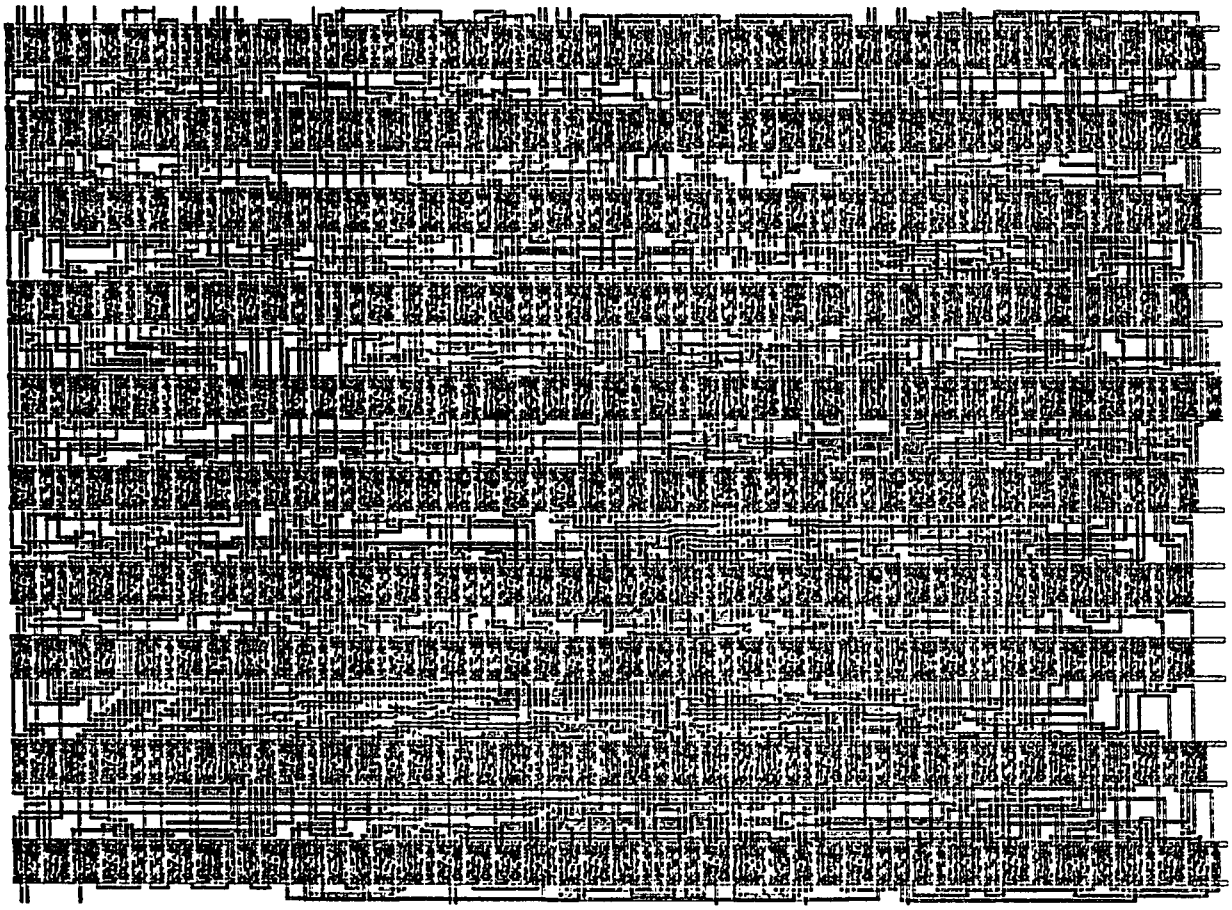

Figure 4.8: Divider layout produced by MAGIC, area = $1200 \times 1640\ \mu m^2$.

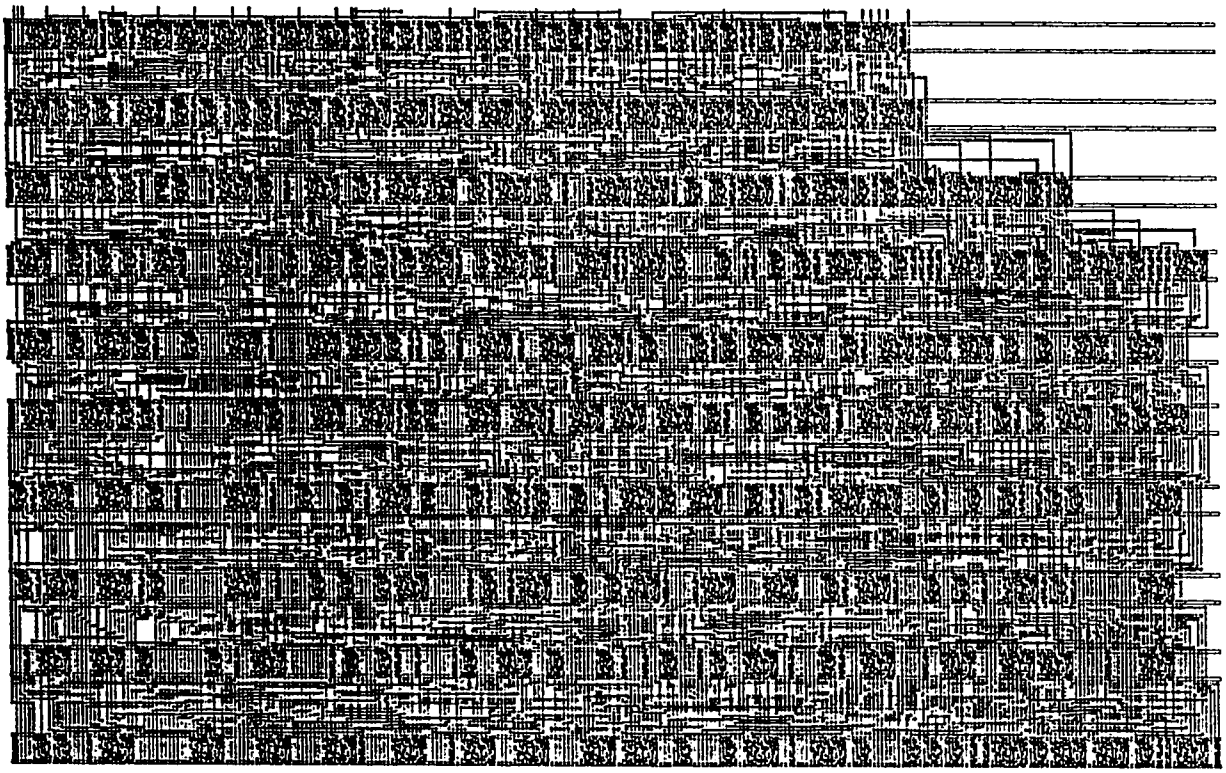

Figure 4.9: Instruction memory layout produced by MAGIC, area = $1440 \times 2328\ \mu m^2$

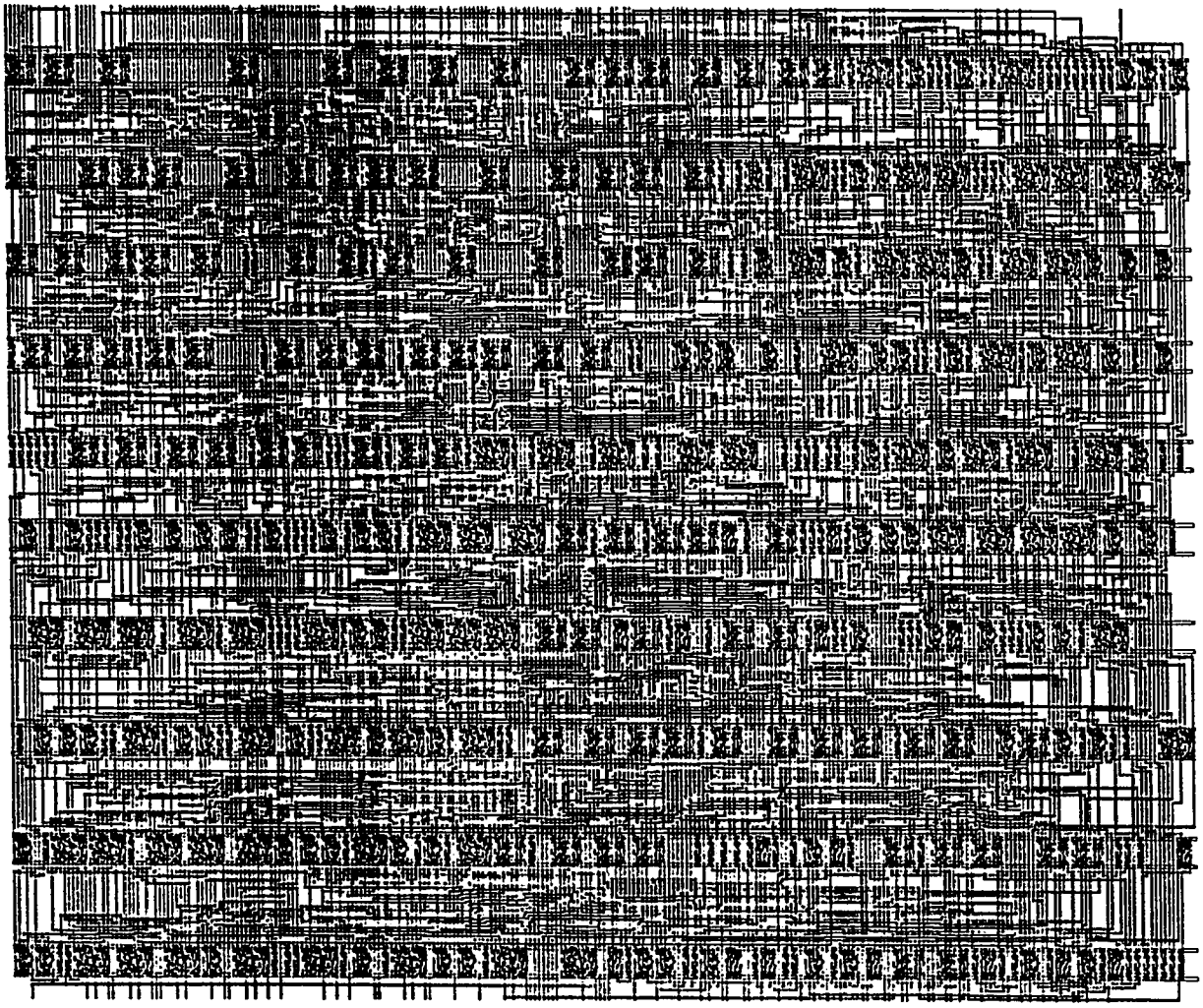

Figure 4.10: Data registers layout produced by MAGIC, area = $1808 \times 2192\ \mu m^2$.

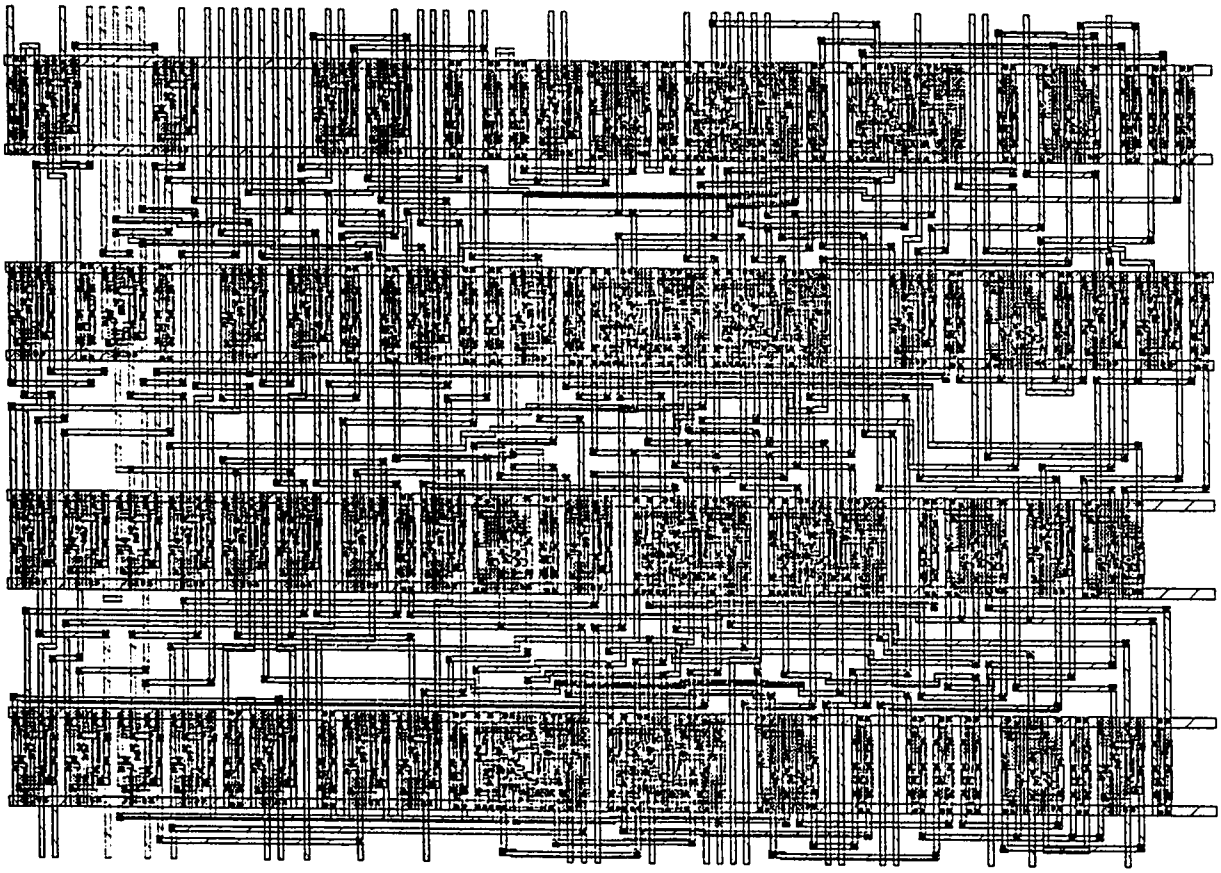

Figure 4.11: Data acknowledgment circuit layout produced by MAGIC.

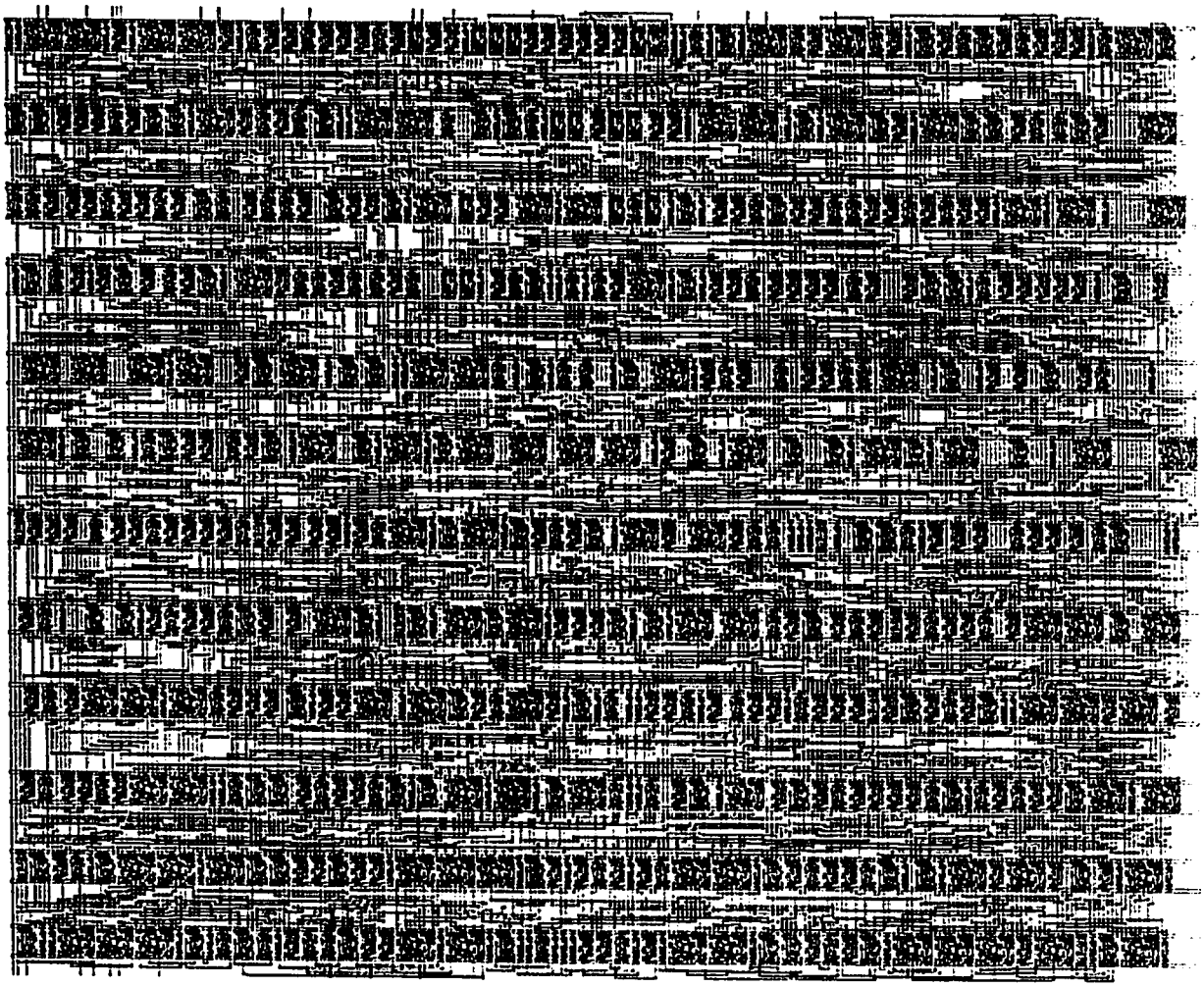

Figure 4.12: FIFO-RAM layout produced by MAGIC, area = $1573 \times 2112\ \mu m^2$.

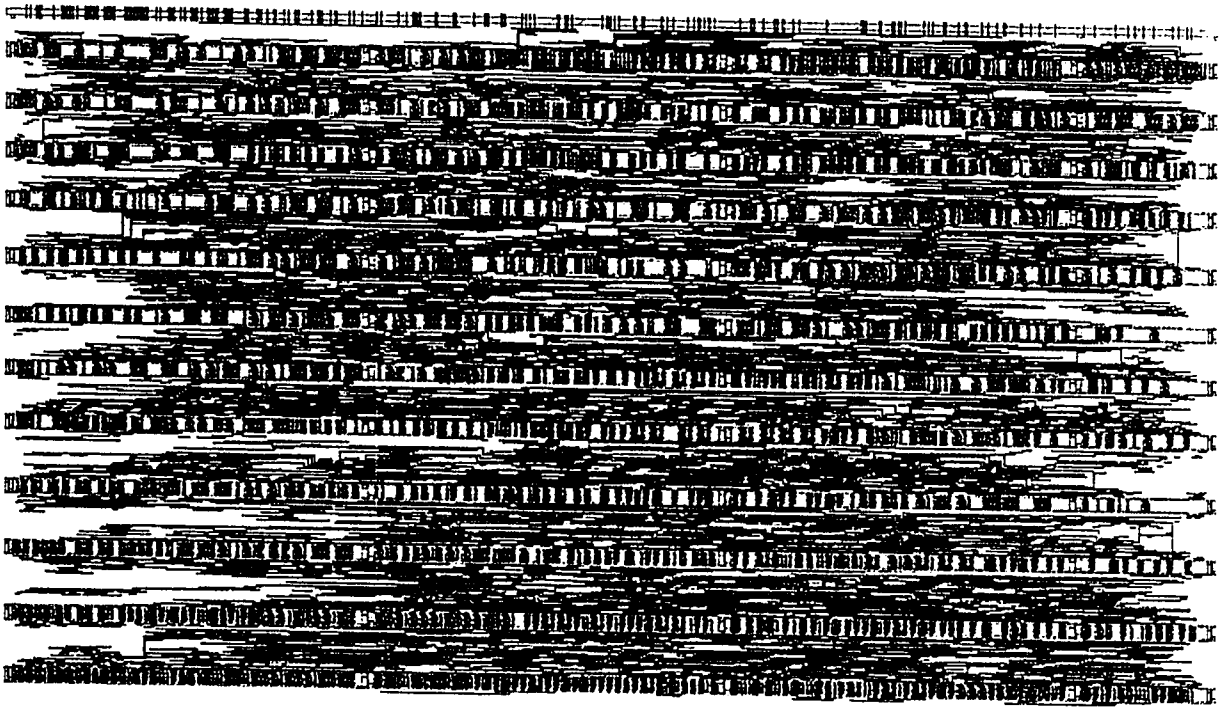

Figure 4.13: Parallel dataflow controller layout produced by MAGIC, area = 3154 × 3840 μm^2 .

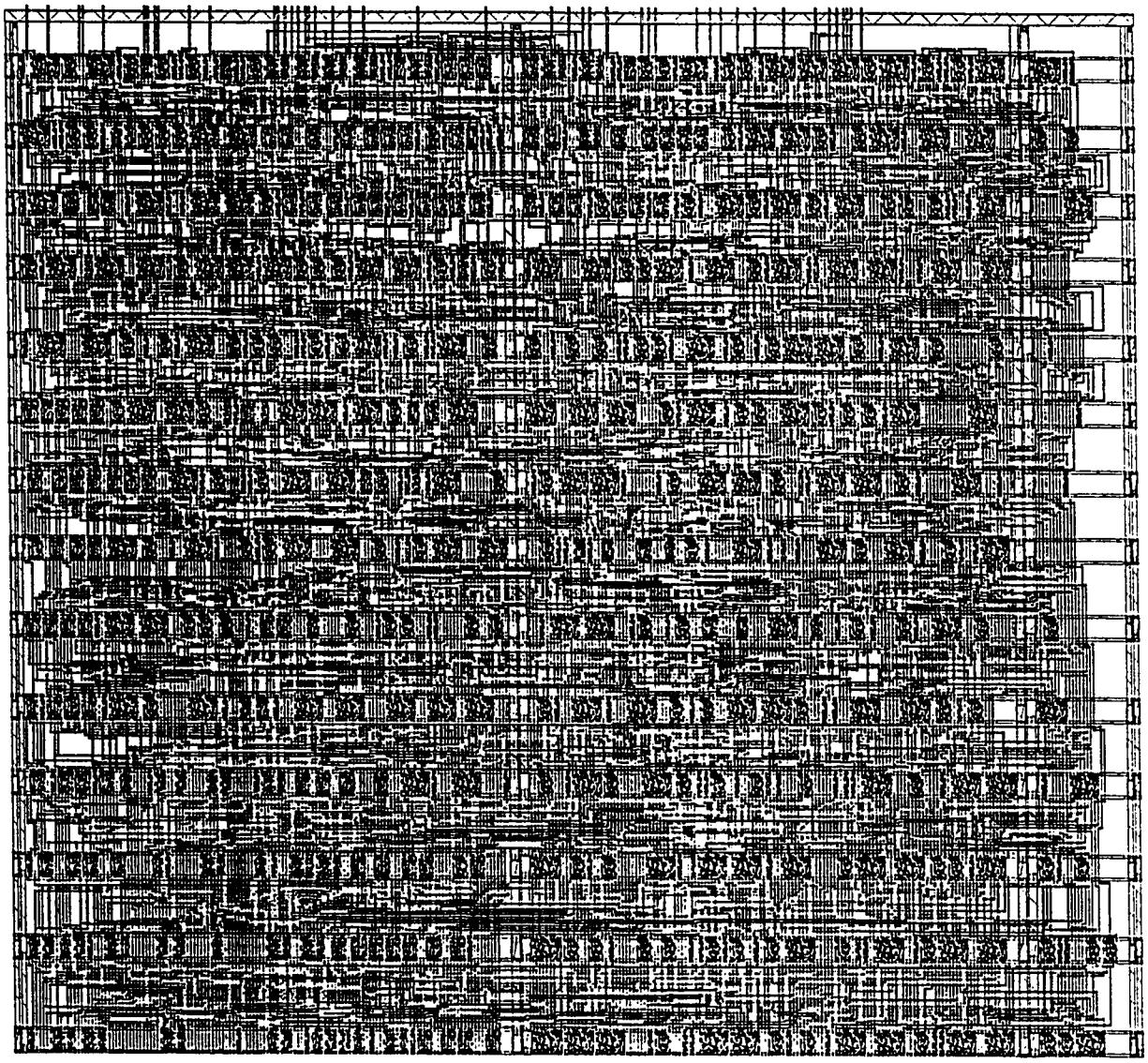

Figure 4.14: Matching unit layout produced by MAGIC, area = 2381 × 2608 μm^2 .

4.5 Program Execution

No commercially available/proposed processor is fully suitable for comparing the performance of this processor. However to obtain some quantitative comparisons (even if not completely satisfactory one) this processor has been compared with 80486 von Neumann computer.

To verify the correctness of the design, and determine the operating speed. A simple expression (Equation 4.1) for array multiplication and addition has been mapped on two processing elements.

$$\mathbf{x = (a + b) \times (c + d)} \tag{4.1}$$

where $a = [a_0....a_n]$, $b = [b_0....b_n]$,

$$\begin{bmatrix} c_0 \\ \vdots \\ c_n \end{bmatrix} \begin{bmatrix} d_0 \\ \vdots \\ d_n \end{bmatrix}$$

The mapping of this expression is shown in Figure 4.15. Figure 4.16 shows the instruction level mapping and dataflow among the registers and processing elements. Simulations are performed in a switch level simulator *irsim* (as discussed in Section 4.4.2). The instruction and data memory of processing elements are loaded with the data using the command file of the simulator. The clock of 500 nS ($2Mhz$) is used in simulations. The simulation results are shown in Figure 4.17, which shows that the first result is produced after 24,000 nS and subsequent results are produced after a delay of 8,000 nS. This reveals that the expression $x = (a + b) \times (c + d)$ has been calculated in only four machine

cycles using only two processing elements, this may be further reduced if proper mapping is used.

The same expression has been implemented on 80486 personal computer. The data for a, b, c, and d is stored in the same named arrays and the result is stored in the *prod* array. Table 4.2 shows the program along with the timing in clock cycles [37]. This von Neumann microprocessor requires 32-clock cycles to execute the expression $x = (a + b) \times (c + d)$, for 8-bit operands.

Table 4.2: Program for the execution of expression $x = (a + b) \times (c + d)$ on 80486.

Label	*Instruction*		*clock − cycles*
	MOV	SI, OFFSET a	1
	MOV	CX, 10	1
AGAIN:	MOV	AL, [SI]	1
	ADD	AL, [SI] + [b]	3
	MOV	[SI] + [a+b], AL	1
	MOV	AL, [SI] + [c]	1
	ADD	AL, [SI] + [d]	3
	MUL	[SI] + [a+b]	13-18
	MOV	[SI] + [prod], AL	1
	INC	SI	1
	LOOP	AGAIN	6-7

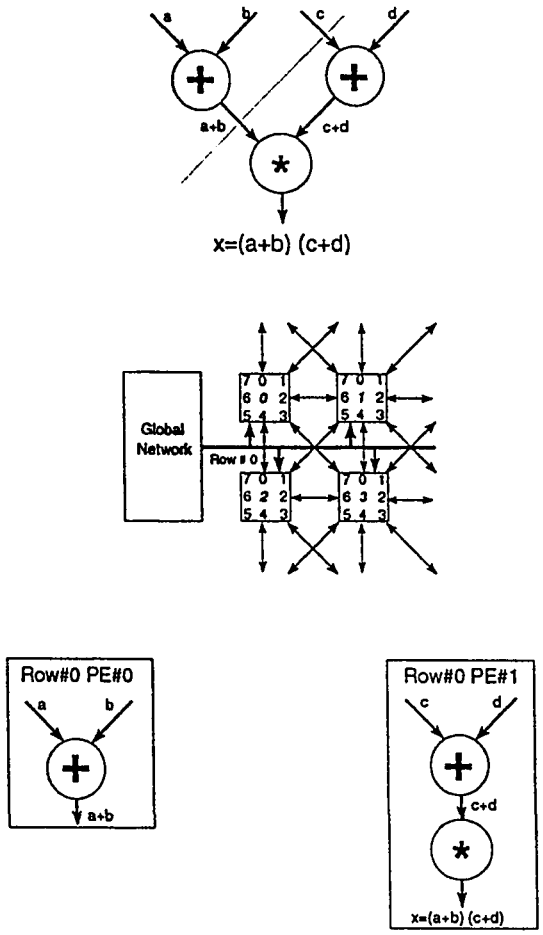

Figure 4.15: Mapping of the expression $x = (a + b) \times (c + d)$ on two processing elements.

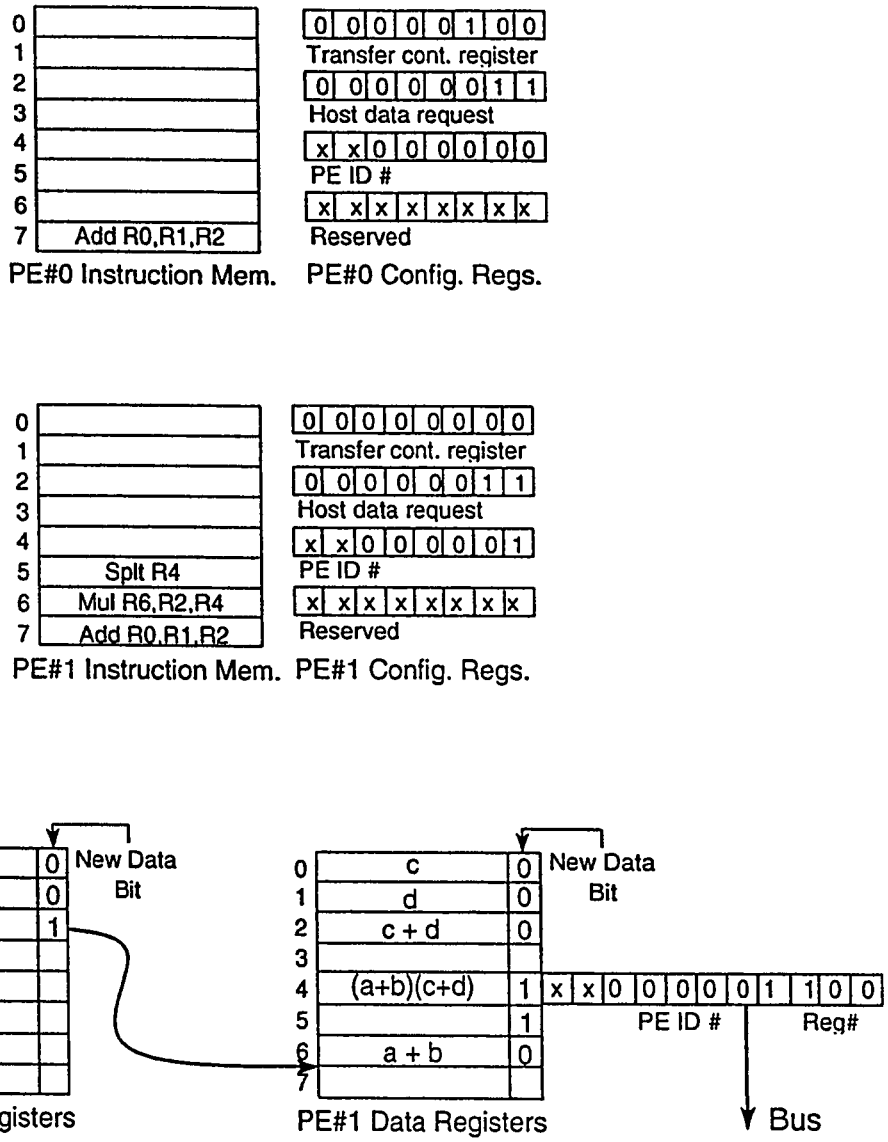

Figure 4.16: Instruction level mapping of the expression $x = (a+b) \times (c+d)$ on the designed processing elements array.

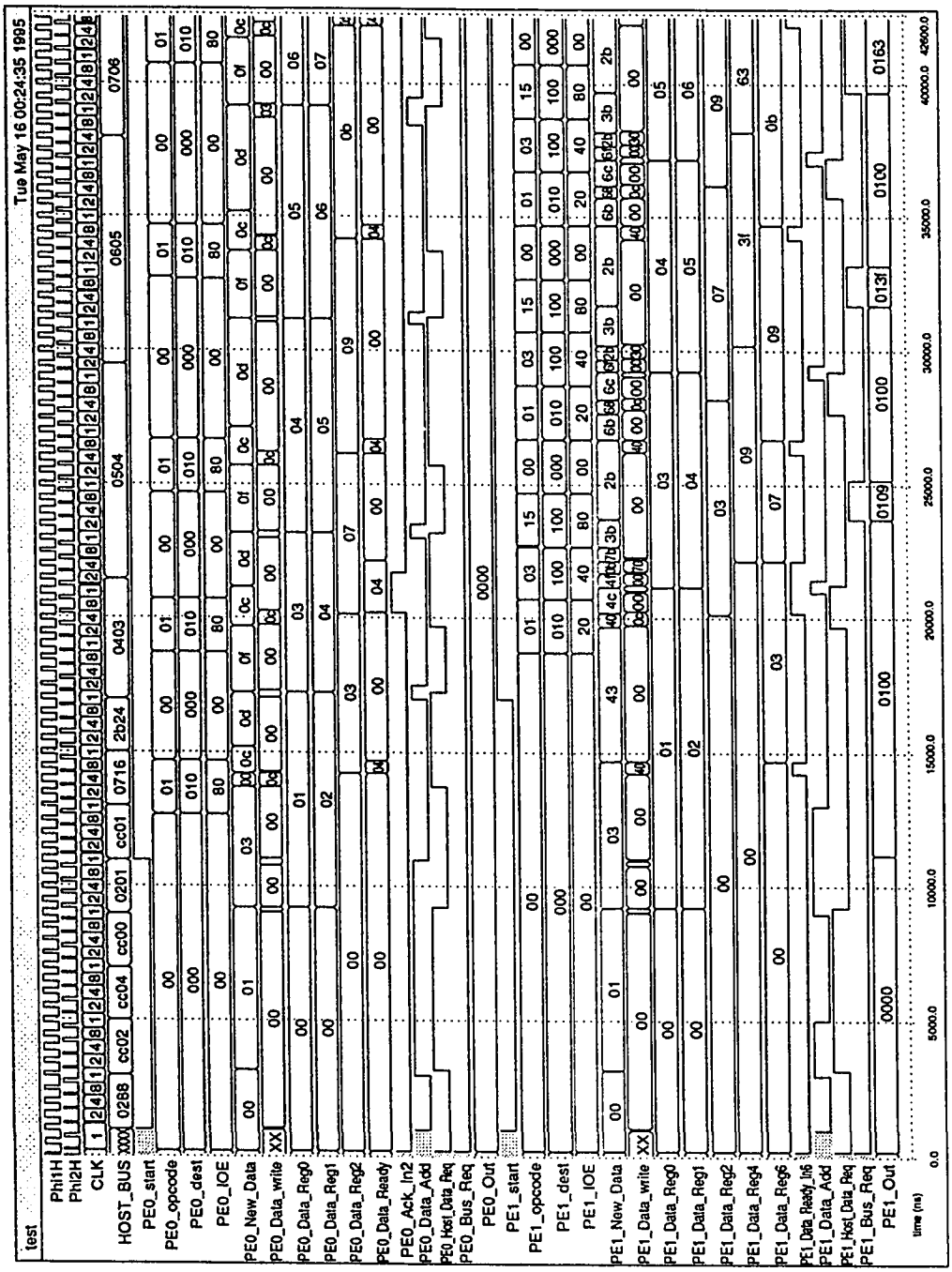

Figure 4.17: *Irsim* simulation results for the expression $x = (a + b) \times (c + d)$.

Chapter 5

Conclusions and Future Research

A Dataflow -RISC-Array processor architecture for use in a wide variety of data and signal processing applications has been designed and implemented in VLSI. A variation of elementary dataflow principles of execution have been introduced, so as to attune the model of execution to the requirements of the RISC architecture for maximal performance.

Since the processor has been designed as a single chip computer, which can be easily programmed for arbitrary algorithms, particular design considerations have been taken to realize high performance. In order to execute single cycle division and multiplication, specialized hardware multiplier [20] and dividers have been designed, with low $area \times time$ ratio instead of software routines as normally done in RISC processors.

The processor is highly reconfigurable and special attention has been given to array topology design and communication among processing elements. This topology gives maximum flexibility to the programmer and results in tighter coupling and faster communication among processing elements. Due to high inter connectivity and reconfiguration it is possible to implement any dataflow graph on this processor array. The topology can be reconfigured to star, pyramid, or binary tree, depending on the structure of the algorithm.

The functionality of the processor has been tested (on all levels of the design), for a simple expression, using machine code of the processor assembly language. This implementation of a dataflow processor will provide a test bench at KFUPM for the future work in the field of data-flow computing. Some of the areas of further research are listed below:

1. Compiler writing and algorithm mapping techniques must be developed to get full advantage of dataflow computing.

2. A large variety of programs (for parallel computing) need to be simulated on the processor.

3. The performance of the processor needs to be compared against other architectures using similar level of technology, especially von Neumann architectures and various multiprocessor architectures such as macro dataflow P-RISC and array processor by Silberman [3].

4. More complex ALU needs to be considered, especially one that includes floating point capabilities.

5. More sophisticated VLSI implementation using handcrafted cells for area minimization.

Bibliography

[1] J. A. B. Fortes and B. W. Wah. Special issue on systolic arrays - from concept to implementation. *IEEE Computer*, 7(20):12–17, July 1987.

[2] S. Y. Kung, S. C. Lo, S. N. Jean, and L. N. Hwang. Wavefront array processors concept to implementation. *IEEE Computer*, 7(20):18–33, July 1987.

[3] S. Weiss, I. Y. Spillinger, and G. M. Silberman. Architectural improvements for a data-driven VLSI processing array. *Journal of Parallel and Distributed processing*, 19:308–322, December 1993.

[4] J. B. Dennis. First version of a dataflow procedure language. *MAC Technical Memorandum, MIT, Cambridge*, 61, 1973.

[5] Arvind and R. S. Nikhil. Executing a program on the MIT tagged token dataflow architecture. *IEEE Transactions on Computers*, pages 1–29, 1989.

[6] J. B. Dennis, W. Y-P. Lim, and W. B. Ackerman. The MIT dataflow engineering model. *Proceedings of the IFIP 9^{th} World Computer Conference*, pages 553–560, September 1983.

[7] G. M. Papadopoulos. Implementation of a general purpose dataflow multiprocessor. *Technical report,MIT laboratory for computer science, Cambrige, MA*, (TR-432), August 1988.

[8] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 1(28):34–52, January 1985.

[9] T. Yua, T. Shimada, K. Hiraki, and H. Kashiwagi. Sigma-1: A dataflow computer for scientific computations. *Proceedings of the* 2^{nd} *International Conference on vector and parallel processors in computational science*, pages 28–31, August 1984.

[10] I. Koren, B. Mendelson, I. Peled, and G. M. Silberman. A data-driven VLSI array for arbitrary algorithms. *IEEE Computer*, 10(21):30–43, October 1988.

[11] A. S. Tanenbaum. Structured Computer Organization. *Prentice Hall*, 1990.

[12] J. B. Dennis. Dataflow supercomputer. *IEEE Computer*, 13:48–56, November 1980.

[13] B. Randell and P. C. Treleaven. VLSI Architectures. *Prentice Hall*, 1983.

[14] Kai Hwang and Faye' A. Briggs. Computer architecture and parallel processing. *McGraw-Hill*, 1984.

[15] Shinji Komori, Kenji Shima. Souichi Miyata, and Hiroaki Terada. Data Driven Microprocessor. *IEEE Micro*, pages 45–59, June 1989.

[16] Ben Lee and A. R. Hurson. Dataflow Architectures and Multithreading. *IEEE Computer*, 27(8):27–39, August 1994.

[17] J. Gaudiot and Lubomir Bic. Advanced topics in dataflow computing. *Prentice Hall*, 1991.

[18] Richard S. Piepho and William S. Wu. A comparison of RISC architectures. *IEEE-Micro*, pages 51–62, August 1989.

[19] Robert P. Colwell, Charles Y. Hitchcock III, E. Douglas Jensen, H. M. Brinkley Sprunt, and Charles P. Kollar. Computers, complexity, and controversy. *IEEE-Computer*, 18:9–19, September 1985.

[20] Sadiq M. Sait, Aamir. A. Farooqui, and Gerhard. F. Beckhoff. A Novel Technique for Fast Multiplication. *IEEE Phoenix Conference on Computers and Communication*, March 1995.

[21] Kai Hwang. Computer Arithmetic: Principles, Architecture, and Design. *John Wiley*, 1979.

[22] John P. Hayes. Computer architecture and organization. *McGraw-Hill*, 1978.

[23] K. Kozminski. OASIS: Open Artchitecture Silicon Implementation System Users Guide. *MCNC, Research Triangle Park, North Carolina*, October 1992.

[24] H. H. Guild. Some cellular logic arrays for non-restoring binary division. *Radio Electron. Eng.*, 39:345–348, 1970.

[25] J. C. Majithia. Nonrestoring binary division using a cellular array. *Electronic Letters*, 6:303–304, 1970.

[26] Maurus Cappa and V. Carl Hamacher. An augmented iterative array for high speed binary division. *IEEE Transactions on Computers*, c-22(2):172–175, February 1973.

[27] Jean-Luc Gaudiot, Rex W. Vedder, George K. Tucker, Dennis Finn, and Michael L. Campbell. A Distributed VLSI Architecture for efficient signal and data processing. *IEEE Transaction on Computers*, c-34(12):1072–1087, December 1985.

[28] V. G. Grafe and J. E. Hoch. The epsilon-2 multiprocessor system. *Journal of Parallel and Distributed processing*, 10:309–318, 1990.

[29] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing. *Proc.* 16^{th} *Int'l Symp. Computer Architecture, IEEE CS Press.*, pages 262–272, 1989.

[30] Albert Paul Malvino. Digital computer electronics: An introduction to microcomputers. *McGraw-Hill*, 2^{nd} Edition, 1983.

[31] S. B. Akero, D. Harel, and B. Krishnamurthy. The star graph: An attractive alternative to the n-cube. *Proceedings of the 1987 International Conference on Parallel Processing*, pages 393–400, August 1987.

[32] Zainalabedin Navabi. VHDL Analysis and Modeling of Digital Systems. *McGraw-Hill*, 1993.

[33] R. Lipsett, C. Schaefer, and C. Ussery. VHDL: Hardware Description and Design. *Kluwer Academic Publishers*, 1987.

[34] Steve Carlson. Instruction to HDL-Based design using VHDL. *Synopsys Inc.*, 1991.

[35] J. K. Ousterhout et al. MAGIC: A VLSI Layout System. *Proceedings of 21st Design Automation Conference*, pages 152–159, 1984.

[36] Robert N. Mayo et al. 1990 DECWRL/Livermore Magic Release, Digital Western Research Laboratory,. September 1990.

[37] 486SX microprocessor, 487SX math co-processor. *Intel Corporation*, 1991.

Vitae

- Aamir Alam Farooqui
- Born in 1969 at Lahore, Pakistan.
- Received Bachelor of Engineering (B.E.) degree in Electronics Engineering from N.E.D. University of Engineering and Technology, Karachi, Pakistan in 1991. With High Honor's (93.3%), top 2 % of the undergraduate class.
- Worked as an Electronic Design Engineer in Pakistan Steel Mills from 1992 to 1993. During service at Pakistan Steel Mills designed and fabricated microcontroller based, highly sophisticated Instrumentation and Control Systems.
- Joined Electrical Engineering Department of KFUPM as a Research Assistant in September 1993.
- Received Master of Science (M.S.) degree in Electrical Engineering from KFUPM, Saudi Arabia in June 1995.